Arquitetura Orientada a Eventos

Soluções escaláveis e em tempo real com EDA

Casa do Código | alura

ROBERTO PICANÇO

Sumário

- ISBN
- Sobre o livro
- Estrutura do livro
- Sobre o autor
- Agradecimentos
- Prefácio por Robisson Oliveira
- Parte I: Fundamentos
  - 1 Por que Arquitetura orientada a eventos?
  - 2 Entendendo os principais conceitos
  - 3 Padrões de arquitetura para a Arquitetura orientada a eventos
  - 4 Identificando eventos de negócio e documentando nossa solução
- Parte II: Broker de eventos
  - 5 Broker de eventos habilita a Arquitetura orientada a eventos
  - 6 Funcionalidades comuns de um broker no fluxo de publicação e consumo de eventos
  - 7 Os tipos de brokers de eventos
- Parte III: Arquitetura orientada a eventos e além
  - 8 Padrões arquiteturais que se beneficiam da Arquitetura orientada a eventos
  - 9 Arquitetura orientada a eventos no contexto de uma plataforma de integração
- Parte IV: Execução dos projetos de EDA
  - 10 Projeto 1 – Plataforma de e-commerce
  - 11 Projeto 2 – Envio de notificações de eventos para parceiros
  - 12 Projeto 3 – Rastreamento da frota de caminhões em tempo real
  - 13 Bibliografia

ISBN

Impresso: 978-85-5519-351-4

Digital: 978-85-5519-350-7

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Picanço, Roberto
Arquitetura orientada a eventos : soluções escaláveis e em tempo real com EDA / Roberto Picanço. -- 1. ed. -- São Paulo : AOVS Sistemas de Informática, 2023.

ISBN 978-85-5519-351-4

1. Ciência da computação 2. Processamento de dados 3. Software de aplicação 4. Tecnologia da informação I. Título.

23-173456 CDD-004.2

Índices para catálogo sistemático:

1. Software : Arquitetura, projeto e desenvolvimento : Especificações técnicas : Ciência da computação 004.2

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o livro

À medida que as organizações começam a ganhar maturidade em sua estratégia de transformação digital, elas passam a depender cada vez mais de soluções digitais. Elas desenvolvem essas soluções através do software para prosperar seus negócios e, consequentemente, se destacar no mercado e sair na frente da concorrência.

Uma das principais metas dessas empresas é reduzir ao máximo o tempo entre a ideia e a implantação da solução em produção, sem abrir mão da qualidade, e gerar valor para seus clientes o mais rápido possível — este é o bom e velho *time-to-market*.

A direção que as organizações estão seguindo para acelerar o *time-to-market* é dividir para conquistar, passando de um modelo centralizado para um modelo distribuído, seja de times, processos e tecnologias.

Quando criamos novas soluções ou migramos nossos sistemas monolíticos em pequenas aplicações distribuídas para ganhar velocidade e agilidade, estamos inserindo uma grande complexidade na nossa solução digital. Em vez de termos um único sistema com simples chamadas internas de métodos para atingir um objetivo de negócio, passamos a ter várias aplicações independentes se comunicando através da rede, tendo que lidar com serviços remotos indisponíveis ou com alta latência.

A arquitetura orientada a eventos veio para mitigar essa complexidade do modelo distribuído, minimizando o acoplamento entre as aplicações, adicionando assincronicidade, processamento em tempo real e flexibilidade para adaptar o negócio às mudanças de forma ágil com mínimo de impacto. As organizações estão começando a entender seus benefícios e querem implementá-la em suas soluções digitais, mas têm uma grande dificuldade devido à falta de familiaridade com os conceitos, padrões e casos de usos que cercam essa arquitetura.

Neste livro, vamos ajudar o leitor e a leitora na sua jornada de implementação da arquitetura orientada a eventos, explorando conceitos-chaves, padrões arquiteturais e as principais capacidades dos *brokers* de eventos, que são considerados o coração dessa arquitetura. O objetivo é responder a pergunta: *Quais os padrões arquiteturais e tipos de brokers de eventos de mercado mais apropriados para o meu caso de uso?* Na parte final, vamos exercitar o que aprendemos e implementar projetos orientados a eventos simulando alguns casos de uso.

Para quem é este livro?

Este livro foi escrito para pessoas de tecnologia com interesse no assunto, líderes de TI, pessoas desenvolvedoras, engenheiras e arquitetas de software, ou qualquer outro papel que precisa do conhecimento teórico e prático dessa arquitetura.

O principal objetivo do livro é fornecer uma base sólida dos conceitos que envolvem a arquitetura orientada a eventos, contribuindo para a tomada de decisão ao adotá-la.

Estrutura do livro

Este livro está dividido em quatro partes:

Parte I – Fundamentos: Entender os fundamentos da arquitetura orientada a eventos nos dará uma base de conhecimento sólida para uma tomada de decisão apropriada de acordo com o problema que queremos resolver. Nessa parte, vamos entender o porquê dessa arquitetura, explorar seus principais conceitos e padrões arquiteturais e de design, bem como seus benefícios e os desafios que vêm junto ao adotá-la.

Parte II – Broker de eventos: O *broker* de eventos habilita a arquitetura orientada a eventos com seu suporte ao modelo *publish-subscribe*. Ele possui capacidades críticas para uma EDA distribuída, confiável e gerenciável. Nesta parte, vamos entender os tipos de *broker* de eventos existentes no mercado e explorar suas principais características e funcionalidades, o que nos possibilitará escolher o *broker* de acordo com nosso caso de uso.

Parte III – Arquitetura orientada a eventos e além: Arquitetura orientada a eventos não é uma solução *stand-alone*. Isso significa que ela é implementada em conjunto com outras arquiteturas e plataformas, como microsserviços, *serverless*, processamento de *streaming*, Internet das Coisas (IoT), inteligência artificial, API Management (APIM), Integration Platform as a Service (iPaaS) e muitas outras. Nessa parte, vamos entender como podemos trabalhar em conjunto com outras arquiteturas e plataformas e os benefícios que são gerados por ela dentro desse contexto.

Parte IV – Execução dos projetos de EDA: Nesta última parte, nosso foco será no desenvolvimento de alguns projetos de EDA para colocar em prática o que aprendemos. Vamos definir o *broker* de eventos e os padrões arquiteturais que usaremos na nossa solução de acordo com o caso de uso.

Essa sequência tem uma ordem lógica de assuntos. Por esse motivo, recomendo a leitura sequencial do livro. Contudo, os capítulos também podem ser lidos isoladamente. Neste caso, recomendo pelo menos a leitura sequencial da *Parte I - Fundamentos*, pois ela dará uma base de conhecimento para as partes seguintes.

Convenções

Como existem muitos termos em inglês relevantes, optamos por traduzi-los para português e colocar sua versão original em inglês na sua primeira ocorrência entre parênteses. Por exemplo: *broker* de eventos (*event broker*). Pode ocorrer de alguns termos não terem uma tradução adequada e/ou terem seu nome original já consolidado — nesses casos, deixaremos em inglês, como *query* e *time-to-market*.

> **Código-fonte**
>
> Todo o código-fonte das aplicações e os arquivos criados no livro estão disponíveis no GitHub, no seguinte repositório: https://github.com/rpicanco/livro-eda.

Sobre o autor

Roberto Picanço é um engenheiro de software desde 2007. Tem uma sólida experiência em integração de sistemas e sistemas distribuídos como desenvolvedor e arquiteto de soluções. Roberto atuou como consultor e especialista de TI em uma variedade de setores da indústria, tais como financeiro, aviação, telecomunicação, energia e mídia. Desde 2018, ele tem ajudado empresas no desenvolvimento de aplicações modernas utilizando DDD, BDD, automação, microsserviços, arquitetura orientada a eventos (EDA) e computação em nuvem. Além disso, Roberto tem experiência em gerenciamento de produto, com foco em produtos relacionados a EDA e Integration Platform as a Service (*iPaaS*).

Agradecimentos

Gostaria de agradecer a Deus e aos meus familiares por todo suporte. Em especial, à minha esposa e parceira, Manuela, por todo amor, carinho e paciência, principalmente neste longo período de escrita. À minha filha, Júlia (*In memoriam*), que sempre estará em meu coração. Aos meus pais, Cecília e Ismael, que tanto amo.

E não poderia faltar a minha cachorrinha, Mel, que esteve sempre disposta a aquecer meus pés nesse clima frio da região serrana do Rio de Janeiro.

Agradeço aos meus colegas e excelentes profissionais que me ajudaram com feedbacks e insights:

- Robisson Oliveira – Prefácio e revisão técnica;
- Leonardo Américo – Revisão técnica;
- Thiago Ferreira – Implementação do projeto do capítulo 10.

Não posso deixar de agradecer à equipe da Casa do Código pela oportunidade e a todas as pessoas com quem tive o prazer de trabalhar e que, ao longo desse tempo, contribuíram para o meu crescimento profissional.

Prefácio por Robisson Oliveira

Ainda me lembro da primeira vez que trabalhei com uma Arquitetura Orientada a Eventos, embora nem soubesse na época que tinha esse nome. Para mim aconteceu tarde, posso dizer: dos 15 anos de experiência em desenvolvimento de software, aconteceu no meu sétimo ano.

Um usuário fazia um cadastro em um sistema, depois era necessário enviar um e-mail de notificação para dezenas (às vezes centenas) de destinatários, e lá estava o usuário esperando o cadastro e mais todos esses envios de e-mail para receber um "cadastro efetuado com sucesso!", ou pior, às vezes esperando vários segundos (ou até minutos) para receber uma mensagem de erro e ter que tentar novamente. Então, por acaso – nem me lembro ao certo como –, eu conheci um *broker* de eventos. Agora um pouco reformulado, o usuário fazia o mesmo cadastro, um evento era gerado, o usuário recebia sua mensagem de sucesso quase instantaneamente e um processo em paralelo orientado por esse evento fazia todos os envios de e-mail.

Para mim, isso foi um ponto de inflexão: como eu podia ter desenvolvido software todos esses anos e não ter usado esse tipo de lógica, arquitetura e implementação? O nosso mundo, fora da tecnologia que nos cerca, é totalmente orientado a eventos:

- Você pega o celular e pede um Uber porque precisa se deslocar;
- Termina uma tarefa no seu trabalho e comunica a alguém, que faz algo com essa informação;
- Ao entrar em sua casa, você acende a luz;
- Atualiza o seu perfil da rede social;
- Termina de comer em um restaurante e sinaliza o garçom de que gostaria de pagar a conta.

Todos esses casos, que, para exemplificação, são bem simplórios, são eventos que podem ou não ter uma ação ou reação relacionada. Há poucos meses, me dei conta de que meu carro, quando chega a

10 mil km, agenda automaticamente uma revisão com a concessionária. Achei uma experiência surpreendente, diferente da que eu estava acostumado.

Você já deve ter lido em centenas de lugares que o nosso mundo está cada vez mais volátil, incerto, complexo e ambíguo; que a velocidade com que as coisas mudam está cada vez mais rápida. Cada vez mais, é essencial para que os negócios prosperem a agilidade em perceber e reagir às mudanças e ao mercado – mercado este que está cada vez mais distribuído: hoje nós temos times, processos e tecnologias distribuídas, o que deixa tudo ainda mais complexo.

É neste ponto que chegamos ao objetivo deste livro: para ter um negócio ágil e flexível, é necessário que o software desse negócio seja ágil e flexível. Conhecer a arquitetura orientada a eventos é o que vai permitir que aplicações modernas possam capturar informações e responder em tempo real; que possam criar experiências de uso mais amigáveis em um mundo de negócios cada vez mais conectado e centrado no cliente.

A arquitetura orientada a eventos, embora transformadora, está longe de ser simples. Há diversos padrões, arquiteturas, tecnologias e implementações para se chegar ao mesmo resultado. Pergunte a alguém o que é a arquitetura orientada a eventos e veja a diversidade de respostas – ou a pessoa travar. Como toda decisão tecnológica tem seus pontos positivos e negativos, não há uma linha clara entre o certo e o errado. Para piorar o cenário, algumas vezes times e empresas têm que conviver por anos com a decisão que foi tomada até conseguir mudar.

Porém, quando implementada de forma assertiva, a arquitetura orientada a eventos é um poderoso diferencial nos negócios, pois permite reagir rapidamente aos clientes e criar experiências mais amigáveis. Para pessoas desenvolvedoras e arquitetas, permite maior resiliência, escalabilidade e agilidade em momentos em que é preciso mudar de direção. Para quem realmente gosta de trabalhar

com tecnologia, não posso deixar de comentar que é um desafio muito satisfatório e eletrizante. No meu atual trabalho como Cloud Application Architect, gosto sempre de pensar nos desafios que aparecem e propor soluções para serem *serverless first* e *event-driven first*, por acreditar justamente que vão trazer mais agilidade para os negócios e abstrair a parte mais complexa. Foco no "*first*" aqui como uma tentativa, não como uma solução para tudo.

Este livro é um grande acelerador nessa jornada de conhecer e implementar a arquitetura orientada a eventos. Além de introduzir os fundamentos básicos sobre o assunto, conta com um catálogo de padrões, com seus prós e contras de quando utilizar, as principais tecnologias, casos de uso e exemplos de implementação. Gostaria de ter lido este tipo de trabalho no início da minha jornada trabalhando com esses desafios.

Tive a oportunidade e satisfação de trabalhar com o Roberto, e posso afirmar que a habilidade que mais desenvolvi trabalhando lado a lado com ele foi justamente como trabalhar com arquiteturas orientadas a eventos. Como já dito aqui, eu conheci a arquitetura orientada a eventos no meu sétimo ano de profissão, e comecei a trabalhar com o Roberto no meu décimo segundo ano. Espero que você, leitor(a), consiga extrair o máximo do que este conhecimento pode lhe proporcionar.

Robisson Oliveira — Sr. Cloud Application Architect - Amazon Web Services

Parte I: Fundamentos

Capítulo 1
Por que Arquitetura orientada a eventos?

Conforme as organizações adquirem mais maturidade em sua transformação digital, com estratégias para aumentar sua produtividade, melhorar a experiência do cliente e reduzir custos operacionais, para citar alguns dos benefícios, elas passam a depender cada vez mais de soluções digitais. Elas precisam ser eficientes na entrega de suas soluções digitais, através do software, para diminuir, o máximo possível, o tempo entre a ideia e a implantação da solução em produção — o bom e velho *time-to-market*.

Mas como diminuir o *time-to-market* em um mercado, negócio e tecnologia cada vez mais voláteis, incertos, complexos e ambíguos — isto é, em um contexto VUCA (acrônimo das palavras em inglês *Volatility*, *Uncertainty*, *Complexity* e *Ambiguity*; WIKIPEDIA, c2023).

A direção que as organizações estão seguindo para diminuir o *time-to-market* é dividir para conquistar, **saindo de um modelo *centralizado* para um modelo *distribuído***, seja de times, processos e/ou tecnologias. Tudo está ficando cada vez mais granular.

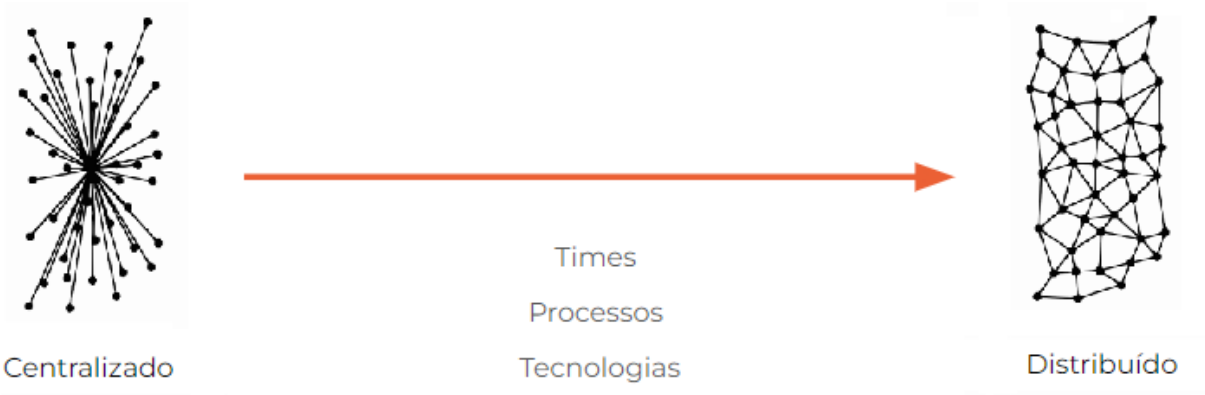

Figura 1.1: Transição do modelo centralizado para o distribuído.

Os times estão sendo divididos por domínio específico do negócio, jornada do cliente ou fluxo de valor, com conhecimentos multidisciplinares e autônomos para aumentar a velocidade. Esses times focam em processos, práticas e ferramentas eficientes no ciclo de vida do desenvolvimento para ganhar agilidade e qualidade na entrega final do software, como a automação dos testes de negócio, segurança e provisionamento da infraestrutura, revisão de código etc.

Em relação à tecnologia, dentro do nosso contexto de software, muitas dessas organizações estão desenvolvendo novas soluções ou migrando seus grandes sistemas monolíticos (centralizado) em várias aplicações distribuídas, que colaboram entre si para atingir um objetivo de negócio e ganhar velocidade e agilidade na entrega da solução.

VELOCIDADE E AGILIDADE

Velocidade: Vá rápido quando você sabe o seu destino;

Agilidade: Vá rápido quando você não sabe o seu destino, mas com a habilidade de mudar a direção rapidamente dadas as incertezas.

Entretanto, ter aplicações distribuídas insere uma grande complexidade na nossa solução digital. Em vez de termos um único sistema com simples chamadas internas de métodos para atingir um objetivo de negócio, passamos a ter várias aplicações independentes se comunicando através da rede, tendo que lidar com serviços remotos indisponíveis ou com alta latência. Nossa solução passa a ser mais difícil de gerenciar, testar, proteger, monitorar, manter e depurar. É aí que entra a **Arquitetura orientada a eventos**, para nos ajudar a mitigar essa complexidade do modelo distribuído.

1.1 O que é exatamente Arquitetura orientada a eventos?

O termo **Arquitetura orientada a eventos** – em inglês, *Event-driven Architecture*, ou simplesmente EDA (utilizarei essa sigla a partir de agora no decorrer do livro) – está sendo usado pelo menos desde meados dos anos 2000 para comunicação entre sistemas através de eventos (FOWLER, 2006).

Antes de explicar o que é EDA, vamos entender o que é *evento* e ver as definições de algumas empresas sobre essa arquitetura.

***Evento*, no contexto de EDA, representa um fato significante, uma mudança de estado; ele é imutável: uma vez que aconteceu, não pode ser desfeito**. Esses eventos podem ser a nível sistêmico ou eventos de negócio — uma tentativa de login do usuário XYZ ou preço do produto XYZ atualizado.

Vamos ver as definições de algumas empresas sobre EDA. Algumas delas estão em inglês, então fiz uma tradução livre.

- Amazon Web Services (AWS, c2023a): "*Uma arquitetura orientada por eventos* ***usa eventos para acionamento e comunicação entre serviços desacoplados*** *e é comum em*

aplicações modernas criadas com microsserviços. Um evento é uma mudança ou uma atualização no estado, como um item adicionado a um carrinho de compras em um site de comércio eletrônico".

- Confluent (AVERY, 2019): "*Arquitetura orientada a eventos (EDA) é um tipo de arquitetura de software que ingere, processa, armazena e* ***reage aos dados em tempo real à medida que são gerados*** (...)".

- Gartner (c2023): "*A Arquitetura orientada a eventos (EDA) é um paradigma de design no qual* ***um componente de software é executado em resposta ao recebimento de uma ou mais notificações de eventos****. EDA é mais fracamente acoplada do que o paradigma cliente/servidor, porque o componente que envia a notificação não conhece a identidade dos componentes receptores no momento da compilação*".

- Google (c2022): "***Uma arquitetura orientada a eventos é um padrão de design de software em que os microsserviços reagem a alterações no estado, chamados de eventos****. Os eventos podem conter um estado (como o preço de um item ou um endereço de entrega) ou podem ser identificadores (uma notificação de que um pedido foi recebido ou enviado, por exemplo)*".

- Microsoft Azure (c2023a): "*Uma arquitetura orientada a eventos consiste em produtores de eventos, que geram um fluxo de eventos, e consumidores de eventos, que escutam os eventos.* ***Os eventos são entregues quase em tempo real, para que os consumidores possam responder imediatamente a eles à medida que ocorrem****. Os produtores são dissociados dos consumidores – um produtor não sabe quais consumidores estão ouvindo*".

- Red Hat (2019): "*A Arquitetura orientada a eventos é um modelo de arquitetura de software para o design de aplicações.* ***Em um sistema orientado a eventos, os componentes de***

captura, comunicação, processamento e persistência de eventos *formam a estrutura básica da solução*".

- Sensedia (2021): *"(...)* ***a capacidade de responder em tempo real a ameaças e oportunidades*** *pode melhorar a experiência do cliente e os resultados do negócio. Isso requer proficiência na propagação, capturando, analisando e reagindo imediatamente a eventos do ecossistema usando arquitetura (EDA) e comunicações assíncronas"* .

- Solace (RIEL, 2022): *"Arquitetura orientada a eventos (EDA) é um padrão de design de software no qual aplicativos desacoplados publicam e subscrevem eventos de forma assíncrona por meio de um broker de eventos.* ***O foco da EDA está na arquitetura em tempo real e pouco acoplamento dos publicadores, brokers de evento e consumidores*** *(...)"*.

Podemos ver algumas palavras que se repetem na definição de EDA no ponto de vista dessas organizações: **microsserviços**, **tempo real**, **reagir/responder imediatamente a eventos**, **alteração/atualização de estado**. Essas palavras-chaves resumem bem a ideia por trás da EDA.

> **ARQUITETURA ORIENTADA A EVENTOS**
>
> É um estilo arquitetural no qual aplicações ou microsserviços reagem a eventos em tempo real após alguma mudança de estado, em vez de responder a comandos.

Quando dizemos **responder a comandos**, estamos nos referindo ao padrão mais tradicional de *request-response*, onde uma aplicação tem que conhecer o *endpoint* e API de outras aplicações para se comunicarem, o que cria um maior acoplamento entre elas.

Figura 1.2: Padrão tradicional de comunicação entre aplicações request-response.

Para entendermos melhor essa diferença entre *comando* e *evento*, nada melhor que uma analogia:

- **Evento**: Entro em uma sala, gero um evento de *entrei na sala* e a luz acende. Esta é uma reação a um evento;
- **Comando**: Entro em uma sala, ligo o interruptor de luz e a luz acende. Este é um comando.

Na analogia do evento, não tenho conhecimento e não peço que as luzes sejam acesas. Em vez disso, um sensor detecta minha presença; ele tem essa responsabilidade. Na analogia do comando, tenho a responsabilidade de saber acender a luz.

A EDA muda a forma como desenvolvemos software. Quando pensamos na complexidade do negócio, nós começamos a pensar em eventos em vez de comandos: pensamos *um evento X ocorreu* em vez de *comando Y deve ser executado*. Essa é uma mudança fundamental de paradigma de como as aplicações são desenvolvidas.

Para exemplificar essa mudança de paradigma, vamos mapear possíveis eventos de negócio e oportunidades ou momentos de negócio que podem ser explorados por uma academia e uma companhia aérea fictícias.

Academia

Eventos

- Aluno João entrou na academia;
- Aluno João iniciou seu exercício;

- Aluno João saiu da academia.

Oportunidades ou momentos de negócio

- Oferecer uma venda adicional para o João com treinamentos fora do seu pacote padrão;
- Notificar o João com sua série de exercícios;
- O professor vai até ao João para orientá-lo;
- João acumula pontos a cada hora de exercício acumulada para receber recompensas.

Companhia aérea

Eventos

- Passageira Maria chegou para o check-in;
- Passageira Maria chegou no seu destino.

Oportunidades ou momentos de negócio

- Oferecer venda adicional para a Maria atualizar sua classe de voo;
- Notificar a Maria com o número do seu portão;
- Oferecer descontos para a Maria na sua próxima viagem por ter alcançado um nível superior no seu programa de pontos;
- Notificar Maria com ofertas de aluguel de automóvel ou caixa eletrônico mais próximo a ela.

No exemplo da academia, eventos gerados pelos alunos, como *Aluno João entrou na academia*, podem gerar oportunidades como *Notificar o João com sua série de exercícios* e *O professor vai até ao João para orientá-lo*. Na companhia aérea, eventos gerados pelos passageiros, como *Passageira Maria chegou no seu destino*, podem gerar oportunidades como *Oferecer descontos para a Maria na sua próxima viagem por ter alcançado um nível superior no seu programa de pontos* e *Notificar Maria com ofertas de aluguel de automóvel ou caixa eletrônico mais próximo a ela*.

Com esses eventos e oportunidades ou momentos de negócio mapeados, usamos a EDA para capturá-los e processá-los em tempo real.

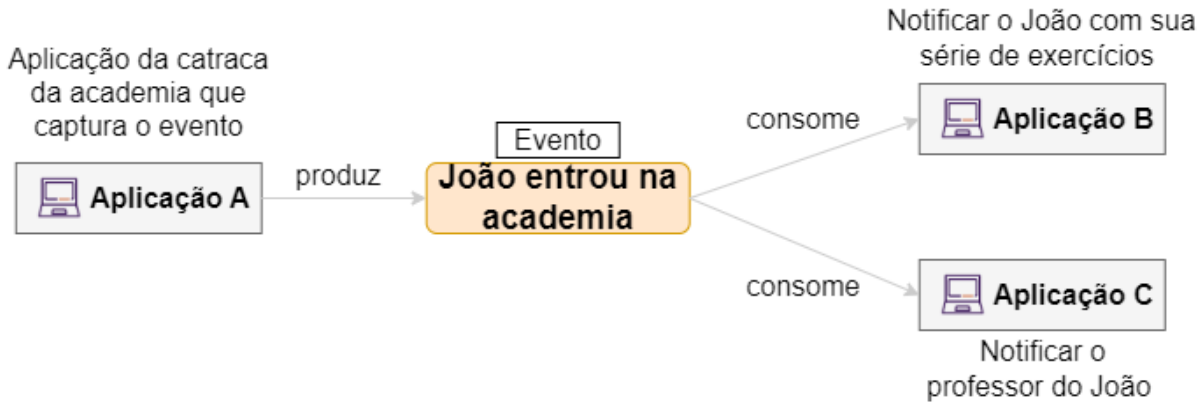

Figura 1.3: Padrão de comunicação através de eventos da academia fictícia.

Eventos de negócio e momentos de negócio

Eventos de negócio: poderíamos chamar EDA de Arquitetura orientada a eventos de negócio. O evento de negócio é o tipo de evento que normalmente queremos capturar e processar para gerar oportunidades e valor para os nossos clientes. Eventos de negócios chegam a todo momento em seu ecossistema através de clientes, fornecedores, parceiros, funcionários etc.

Momentos de negócio: segundo a Gartner, em seu artigo *Embrace Event-Driven Architecture With Responsive Decisions for Digital Business Solutions* (VINCENT; GUPTA, 2017), um momento de negócio é um tipo de evento de negócio, que é uma *oportunidade transitória, explorada dinamicamente*. Significa que ela só faz sentido se processada dentro de um prazo adequado. Imagine um carro envolvido em um acidente. O sistema do carro notifica o hospital mais próximo e o serviço de emergência sem depender de testemunhas. O acidente é um evento que só faz sentido ser processado em um curtíssimo período de tempo (tempo real), podendo salvar vidas. Esse é um bom exemplo de momento de negócio.

Agora que sabemos o que é EDA e as oportunidades de negócio que ela possibilita explorarmos em tempo real, vamos entender seus principais benefícios e desafios que adicionamos à nossa solução ao adotá-la.

1.2 Principais benefícios e desafios envolvidos

Como toda decisão arquitetural, temos que avaliar os prós e os contras de acordo com o problema que queremos resolver. Sistemas que utilizam EDA são *distribuídos*, *assíncronos* e com *consistência eventual*.

COMUNICAÇÃO SÍNCRONA E ASSÍNCRONA

Comunicação síncrona: quando um cliente/aplicação espera receber uma resposta imediata. O cliente/aplicação aguarda a resposta antes de continuar o seu trabalho.

Comunicação assíncrona: quando um cliente/aplicação não tem necessidade de uma resposta imediata. O cliente/aplicação pode continuar o seu trabalho sem ficar *preso* aguardando uma resposta, a qual pode chegar em um determinado tempo no futuro.

Não use EDA para comunicar e gerenciar informações que devem ter consistência forte. Sua natureza distribuída e com comunicação assíncrona significa que seu sistema deve ser capaz de tolerar consistência eventual.

CONSISTÊNCIA EVENTUAL E CONSISTÊNCIA FORTE

Consistência eventual: significa que os dados/informações após uma atualização serão eventualmente consistentes para todos os leitores. Aplicações podem *ler* o dado/informação inconsistente durante um certo período, isso é tolerável.

Consistência forte: significa que os dados/informações imediatamente após uma atualização serão consistentes para todos os leitores. Aplicações não podem *ler* o dado/informação inconsistente, isso não é tolerado. Imagina o impacto em um sistema bancário onde aplicações diversas não *leem* o saldo correto do cliente logo após um saque.

Vamos primeiro entender os principais benefícios da EDA e, na sequência, seus principais desafios.

Benefícios

Baixo acoplamento entre aplicações

O baixo acoplamento significa que as aplicações não precisam saber da existência uma das outras para trabalharem em conjunto. Elas apenas precisam conhecer sobre eventos, que é o único acoplamento para se comunicarem.

Quando falamos de baixo acoplamento entre aplicações, estamos nos referindo ao impacto que uma mudança em uma aplicação terá em outras. Manter as aplicações com um mínimo de acoplamento desencadeia vários outros benefícios na nossa solução de EDA, como *escalabilidade*, *extensibilidade* e *disponibilidade* da solução.

Escalabilidade da solução

É a capacidade do sistema de executar e operar à medida que o número de usuários ou requisições aumenta sem impactar o tempo de resposta. Como as aplicações estão com um mínimo de acoplamento através de eventos, a escalabilidade é facilitada. **O negócio consegue atender ao aumento de usuários sem impactar sua experiência de tempo de resposta.**

Podemos citar dois exemplos de soluções que dificultam a escalabilidade. Primeiro, uma solução com arquitetura monolítica, onde a única opção é escalar o sistema (um conjunto de aplicações fortemente acopladas) por inteiro, o que, na maioria das vezes, se torna inviável e consequentemente o negócio passa a não conseguir atender ao aumento de usuários e requisições. Segundo, uma solução com arquitetura de microsserviços no modelo *request-response*, na qual aplicações estão distribuídas e com pouco acoplamento, porém não de forma minimizada como as da EDA, pois ainda existirá um acoplamento na cadeia de chamadas entre os microsserviços.

Extensibilidade da solução

Extensibilidade da solução é a capacidade de adicionar novas aplicações dentro da solução sem afetar as aplicações existentes.

Esse benefício é poderoso em adaptar o negócio às mudanças de forma ágil com mínimo de impacto.

Um exemplo de solução em que não conseguimos adaptar o negócio às mudanças de forma ágil é uma arquitetura de microsserviços no modelo *request-response*. Imagine uma aplicação A orquestrando diversas chamadas a outras aplicações para atingir um objetivo de negócio. Se adicionarmos uma nova aplicação na solução (adaptar o negócio), precisaríamos alterar a aplicação A para fazer essa nova chamada. Usando EDA, se adicionarmos uma nova aplicação na solução, precisaremos apenas que ela consuma o evento produzido pela aplicação A, sem fazer qualquer alteração nela.

Disponibilidade da solução

A disponibilidade da solução é medida pela porcentagem do tempo que uma solução e suas aplicações ou sistemas estão disponíveis para seus usuários usarem.

Essa medida é feita com os famosos *noves*: quanto maior a quantidade de *noves* na porcentagem, maior é a disponibilidade da solução. Por exemplo, uma solução ou um sistema que tem seu Acordo de Nível de Serviço (Service Level Agreement – SLA) de 99.9% (ou seja, *três noves*) pode ficar até 8 horas e 45 minutos indisponível anualmente. Já uma solução que tem seu SLA de 99.999% (*cinco noves*) pode ficar até 5 minutos indisponível anualmente. Obviamente, quanto mais *noves* de SLA tiver uma solução, mais complexa e cara ela tende a ser. Essa definição vai depender do impacto e da importância da solução para os clientes.

A AWS fornece exemplos interessantes de SLAs de acordo com o tipo da aplicação (confira em https://docs.aws.amazon.com/pt_br/wellarchitected/latest/reliability-pillar/availability.html).

Em relação à EDA, como consequência do baixo acoplamento das aplicações da solução, a disponibilidade é aprimorada, pois essas aplicações com um mínimo de acoplamento **permitem que o**

negócio fique disponível, mesmo se parte dele estiver inoperante. Imagine uma solução de e-commerce em que, mesmo se o componente/aplicação de embalagem e despacho ficar indisponível por um tempo, ainda assim, ela poderá continuar recebendo os pedidos e finalizá-los posteriormente quando o componente voltar a operar.

Processamento em tempo real

Processamento em tempo real é a capacidade da solução de responder a eventos, sejam internos ou externos, em segundos ou milissegundos, permitindo à organização uma rápida tomada de decisão.

Para o usuário em um mundo cada vez mais digital, o comportamento orientado a eventos e a capacidade de resposta em tempo real estão se tornando uma expectativa. **Responder em tempo real a ameaças e oportunidades pode melhorar drasticamente a experiência dos clientes e os resultados do negócio**.

O processamento em tempo real permite a automação da análise dos dados ainda em movimento (termo conhecido em inglês como *Data in motion*), ao invés de esperar os dados serem armazenados (termo conhecido em inglês como *Data at rest*) em algum *storage*. Essa análise de dados em movimento é ideal para problemas como detecção de fraudes, análise preditiva, combate a ameaças de segurança, automação de cadeias de suprimentos, entre outros.

O gráfico a seguir é apenas para ilustrar o evento perdendo seu valor no decorrer do tempo. Quanto mais cedo capturarmos e processarmos o evento, mais valor conseguiremos extrair.

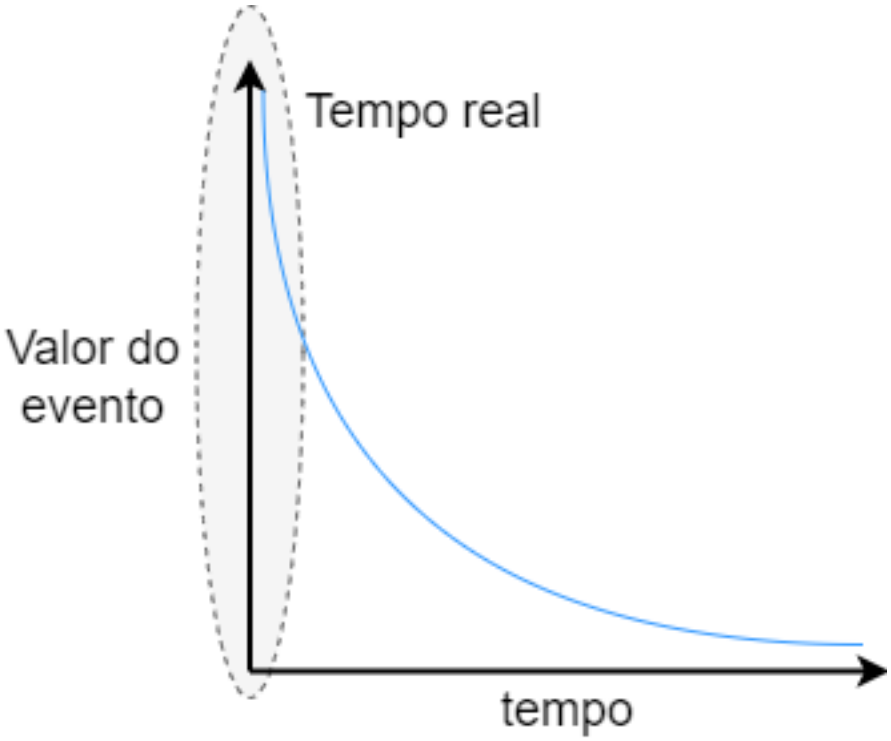

Figura 1.4: O evento perdendo seu valor no decorrer do tempo.

Como nada na vida é perfeito, agora que vimos os principais benefícios da EDA, vamos explorar os principais desafios que essa arquitetura introduz na nossa solução.

Desafios

EDA introduz uma maior complexidade na solução se comparada ao modelo *request-response*

Um fato é que EDA é mais complexa e mais difícil de entender que o modelo *request-response* com o qual já estamos acostumados, no qual a dependência entre aplicações é direta (uma aplicação chama outra através da sua API).

Uma solução EDA é composta por diversas aplicações que se comunicam através de eventos, formando uma cadeia de eventos para atingir um objetivo de negócio. Essa cadeia de eventos é conhecida como **coreografia de eventos**. Defini-la não é intuitivo

e envolve complexidades adicionais, como comunicação assíncrona, fluxo de compensação, rastreabilidade da sequência dos eventos, governança dos eventos, segurança, versionamento dos eventos, esquemas dos eventos etc. Toda essa complexidade será explorada nos próximos capítulos quando falarmos dos padrões relacionados à EDA.

EDA introduz uma maior complexidade no teste da solução

No modelo *request-response*, testar um fluxo de negócio *aplicação A -> aplicação B -> aplicação C -> aplicação D* é mais simples, uma vez que o fluxo do negócio fica mais claro e objetivo; e para cada chamada entre as aplicações, você obtém uma resposta imediata, devido à sua natureza síncrona; faz as devidas validações e segue o fluxo da informação, seguindo a cadeia de chamadas.

Na EDA, testar um fluxo de negócio é mais difícil, pois envolve várias aplicações independentes que se comunicam através da rede e de forma assíncrona, dificultando a rastreabilidade do fluxo e a identificação da origem do problema, o que é agravado com o fluxo de compensação, devido a eventos de erros que *quebram* o caminho feliz da coreografia.

Agora que estão claros os principais benefícios e desafios associados à EDA, vamos entender o que é desenvolvimento de aplicações modernas e qual a sua relação com a EDA.

1.3 Relação com o desenvolvimento de aplicações modernas

Desenvolver aplicações modernas visa habilitar as organizações a inovarem, sem um alto investimento inicial, crescerem e responderem às mudanças rapidamente com práticas e tecnologias

nativas da nuvem e eficientes no ciclo de vida do desenvolvimento e na entrega do software.

EDA não é uma solução *stand-alone*. Isso significa que ela é implementada em conjunto com outros tipos de arquiteturas e soluções, como microsserviços, *serverless*, processamento de *streaming*, Internet das Coisas (IoT), inteligência artificial e muitos outros. **EDA potencializa os benefícios gerados por essas arquiteturas e soluções, que são um dos pilares do desenvolvimento moderno**.

Vamos entender esses pilares na visão de algumas organizações, sem entrar em detalhes, pois não é o objetivo do livro. Mas fica aqui a sugestão de leitura do material disponível nos links a seguir.

Na visão da Microsoft (https://azure.microsoft.com/pt-br/solutions/modern-application-development/#building-blocks), os pilares são:

- Arquitetura nativa da nuvem;
- Interface de usuário alimentada por inteligência artificial;
- Integração usando conectores prontos para uso e gerenciamento de API;
- Bancos de dados totalmente gerenciados;
- Entrega de software usando práticas de DevOps;
- Maximizar a automação nas operações;
- Segurança em várias camadas.

Já a Oracle apresenta um framework baseado em quatro pilares (https://www.oracle.com/br/application-development/modern-app-development/):

- Requisitos principais: requisitos comuns que se aplicam a qualquer aplicativo moderno;

- Princípios de design: práticas recomendadas que regem as arquiteturas de aplicativos;
- Padrões de arquitetura: padrões arquiteturais recomendados;
- Recomendações de tecnologia: uma lista de tecnologias Oracle recomendadas.

A Sensedia tem no seu núcleo a arquitetura moderna – descrita no documento *Microservices Adoption Roadmap* (https://content.sensedia.com/pt-br/microservices-roadmap) –, que tem como pilares:

- A integração e o desenvolvimento moderno;
- A arquitetura de microsserviços, habilitados pela EDA, ou *Event-Driven*.

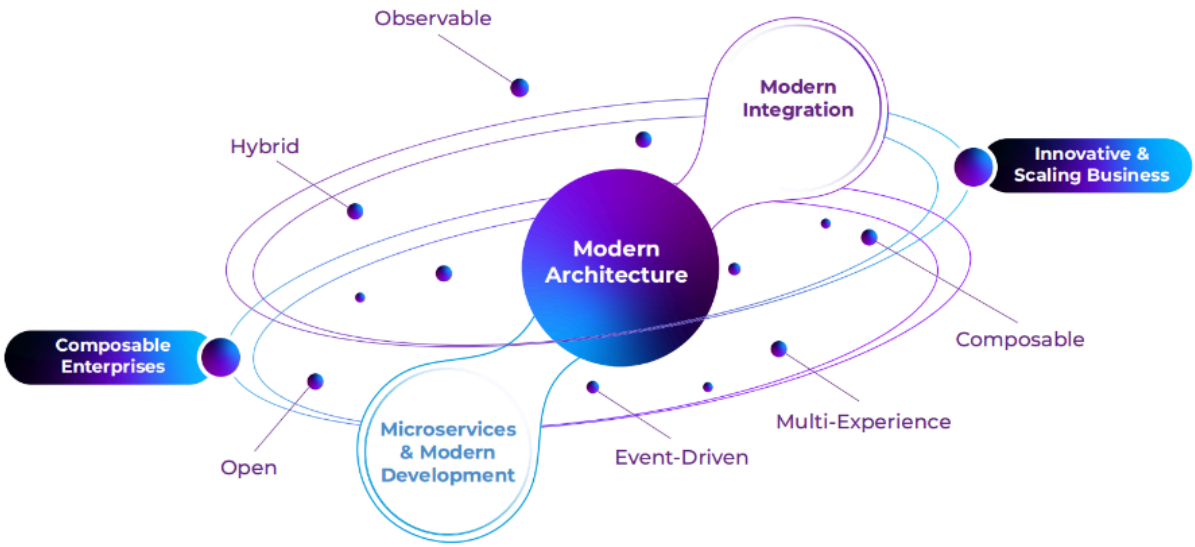

Figura 1.5: Visão Sensedia de uma arquitetura moderna.

Agora que vimos que a EDA é um dos pilares do desenvolvimento de aplicações modernas, através dos seus benefícios, vamos entender, em números, como as organizações estão utilizando EDA e algumas previsões.

1.4 Arquitetura orientada a eventos nas organizações e algumas previsões

Nesta última seção do capítulo, vamos mostrar alguns dados de pesquisas sobre EDA e algumas previsões da sua adoção nas organizações. As pesquisas estão em inglês, então fiz uma tradução livre.

Na pesquisa global *The Great EDA Migration*, feita em novembro de 2021 pela Coleman Parkes e encomendada pela Solace (2021), afirma-se que: "*desde o clique de um pedido até a jornada de um contêiner pelo mundo, as informações do evento nunca foram tão cruciais para as organizações. Não é surpresa que 85% das organizações em todo o mundo tenham se voltado para a EDA*". Segundo ela, há duas razões principais pelas quais as organizações procuram investir em EDA: seus recursos em tempo real, que lhes permitem reagir com mais rapidez e eficiência a um ambiente de negócios acelerado; e sua natureza desacoplada, que garante a capacidade de se adaptar facilmente.

Já em suas previsões, através do relatório *Predictions 2022: Software Development*, a Forrester (2021) afirma que: "*pelo menos um terço das organizações focarão em EDA. Em 2020, 12% dos desenvolvedores globais disseram que sua empresa se concentrava em EDA; isso cresceu para 20% em 2021. A necessidade contínua de abordar eventos de negócios em tempo real expandirá ainda mais: em 2022, esperamos que 35% das organizações de TI tenham um foco principal em EDA*". Já no relatório de 2023, a Forrester (2022) apenas se limitou a dizer que "*Em 2022, a Arquitetura orientada a eventos começou a se firmar*".

O quanto as organizações usam a Arquitetura orientada a eventos?

Segundo a Gartner (2021a), em sua pesquisa *Gartner Predicts 2022: Modernizing Software Development is Key to Digital Transformation*,

feita com líderes de engenharia de software, 64% das organizações atualmente usam EDA em uma ampla variedade de casos de uso ou de forma generalizada.

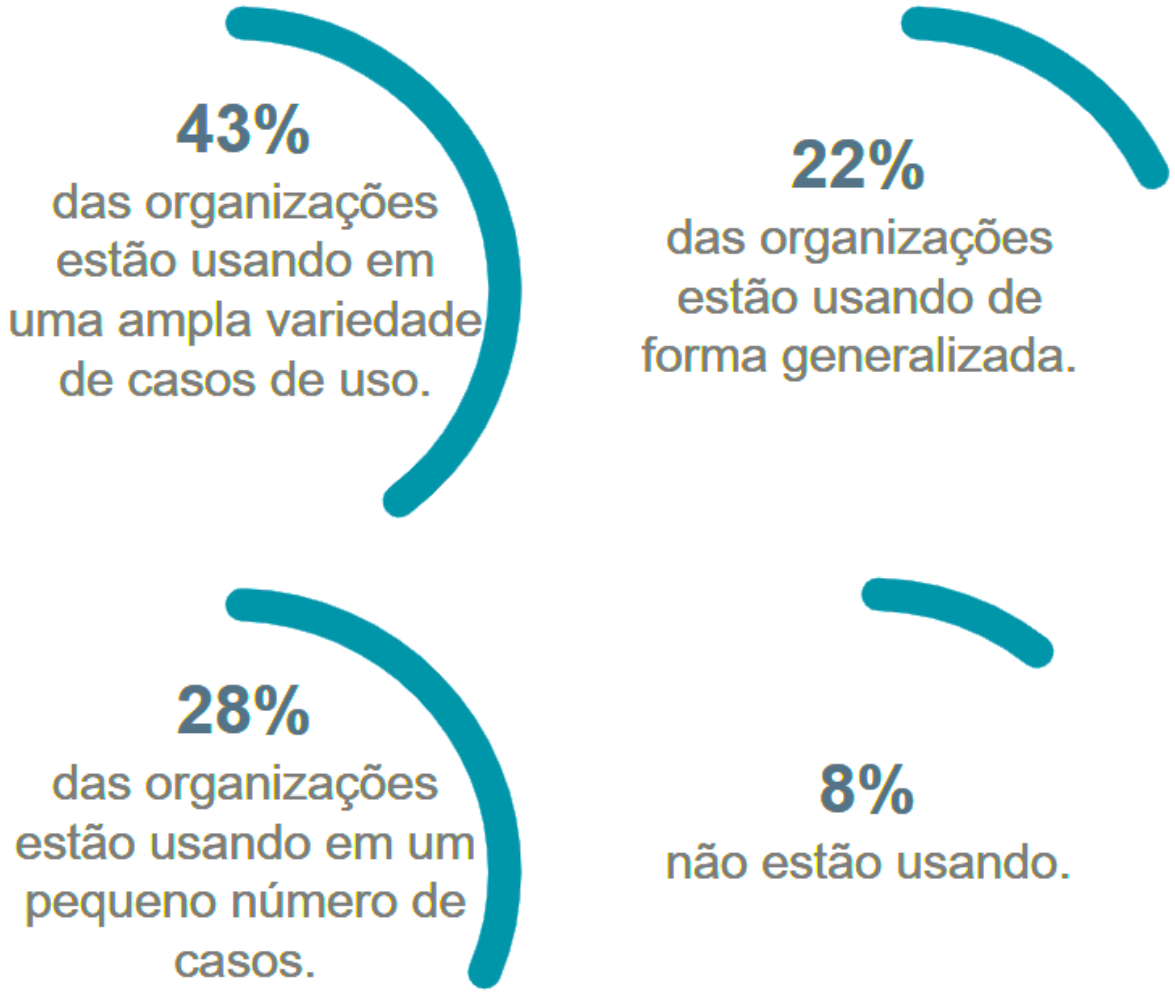

Figura 1.6: Resultado da pesquisa da Gartner sobre o quanto as organizações usam EDA.

A Gartner (2019) afirma, em seu relatório *Maturity Model For Event Driven Architecture*, que: "*como o domínio da EDA é essencial para os negócios digitais e alcançar esse domínio é um processo de várias etapas, as organizações – ainda sem um plano de ação para uso estratégico da EDA – devem começar agora*".

Já a pesquisa mais recente encomendada pela Solace (2023), *Getting in Sync: Unlocking the Exponential Business Value of Real-*

Time Event-Driven Data Flows, realizada pela IDC em dezembro de 2022, revelou que 67% das organizações implementam EDA e que 93% das organizações, que se consideram com maturidade avançada na implementação da EDA, descobriram que a EDA atendeu ou superou as expectativas.

A pesquisa afirma que "*as organizações ganham experiência ao concluir seus primeiros casos de uso, tornando mais fácil ver o potencial e obter suporte para casos de uso adicionais. A maioria das organizações pesquisadas está planejando adicionar 2 ou 3 casos de uso de EDA nos próximos 24 meses para que eles possam fazer uso de todos os novos dados disponíveis para melhorar as decisões mais rapidamente*" (SOLACE, 2023). Ela também afirma que "*EDA é uma estratégia, não um projeto único, e seus benefícios aumentam com o tempo e com casos de uso adicionais*".

As próximas quatro categorias foram realizadas pela mesma pesquisa da IDC encomendada pela Solace, que contou com 316 respondentes, de 10 países diferentes, com cargos de C-level a arquiteto(a) de TI. Vale a pena fazer uma leitura mais detalhada deste relatório.

Os benefícios e desafios técnicos e para o negócio com a EDA

A pesquisa da Solace (2023) revela os cinco maiores *benefícios técnicos* que as empresas entrevistadas perceberam ao implementar a EDA:

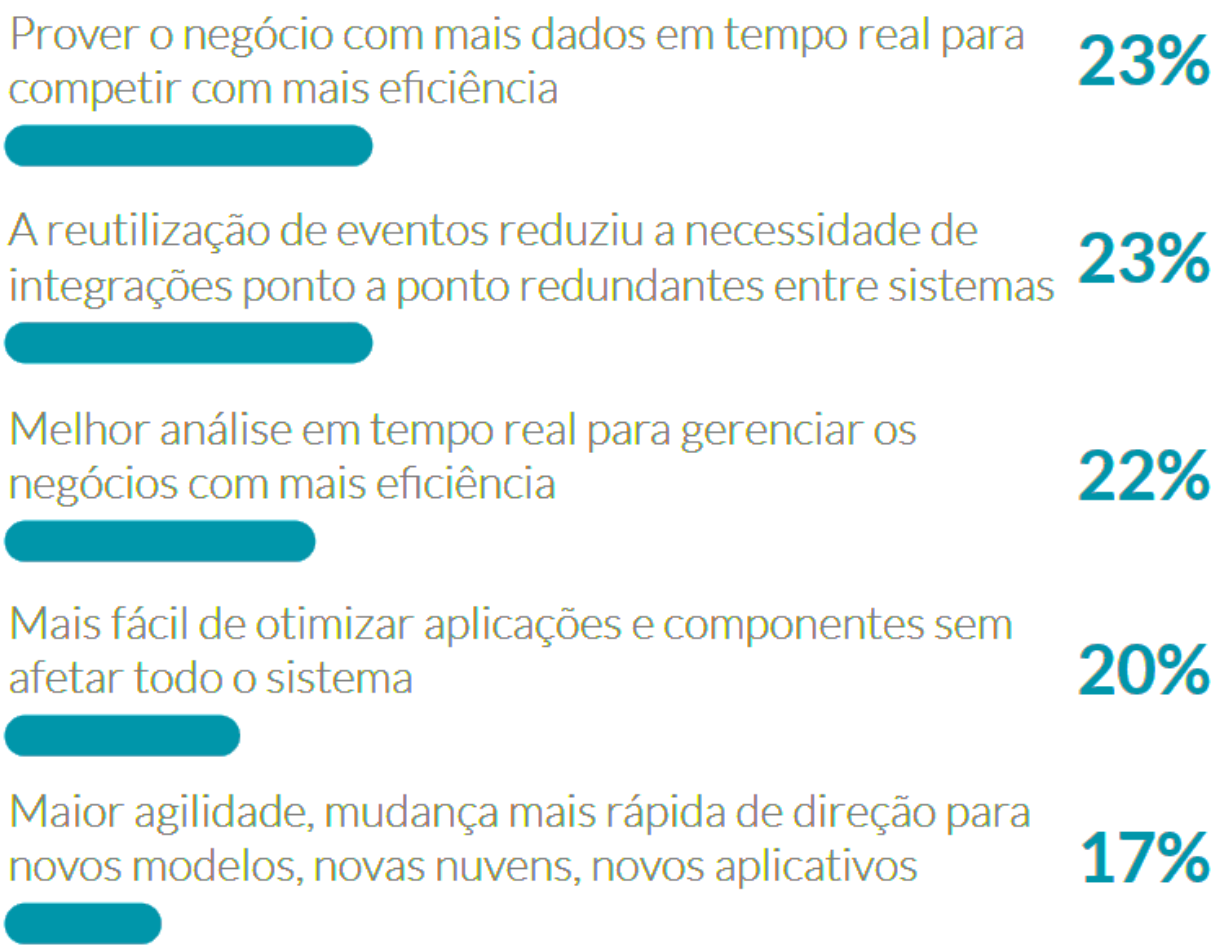

Figura 1.7: Resultado da pesquisa encomendada pela Solace – Os benefícios técnicos com a EDA. Fonte: SOLACE, 2023.

A pesquisa da Solace (2023) também revela os cinco maiores *benefícios para o negócio* que as empresas entrevistadas perceberam ao implementar a EDA:

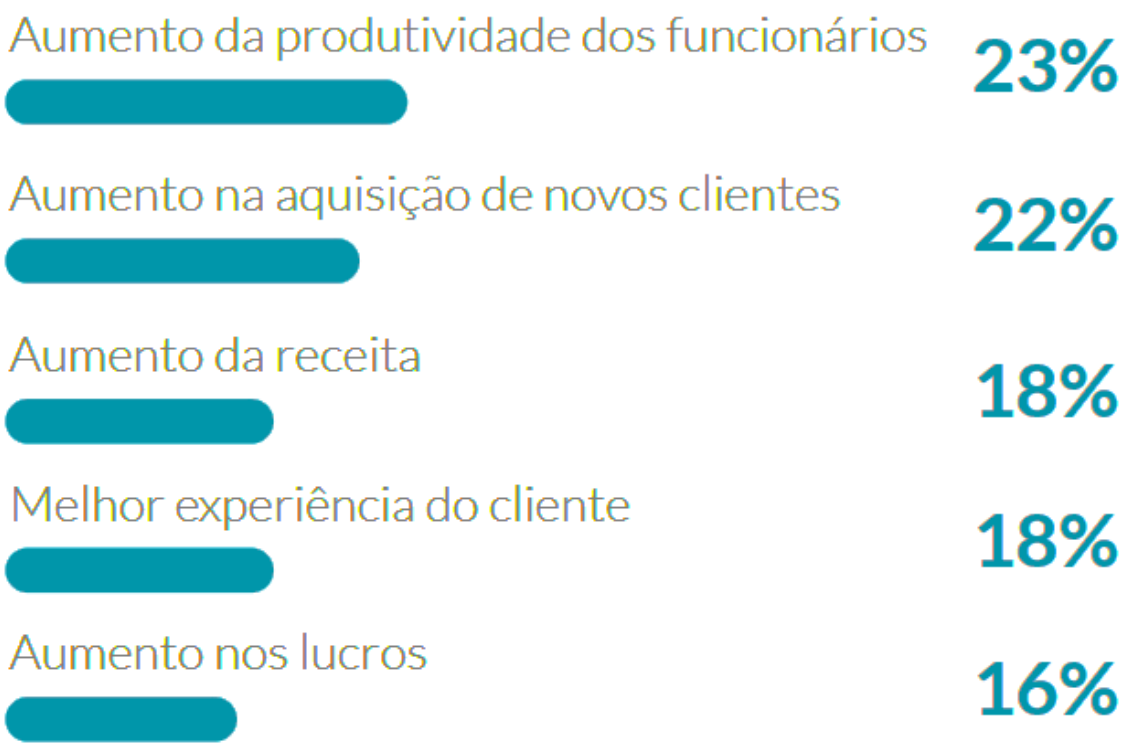

Figura 1.8: Resultado da pesquisa encomendada pela Solace – Os benefícios para o negócio com a EDA. Fonte: SOLACE, 2023.

Já as duas próximas imagens mostram os principais *desafios técnicos e organizacionais* enfrentados pelas empresas que estão implementando a EDA:

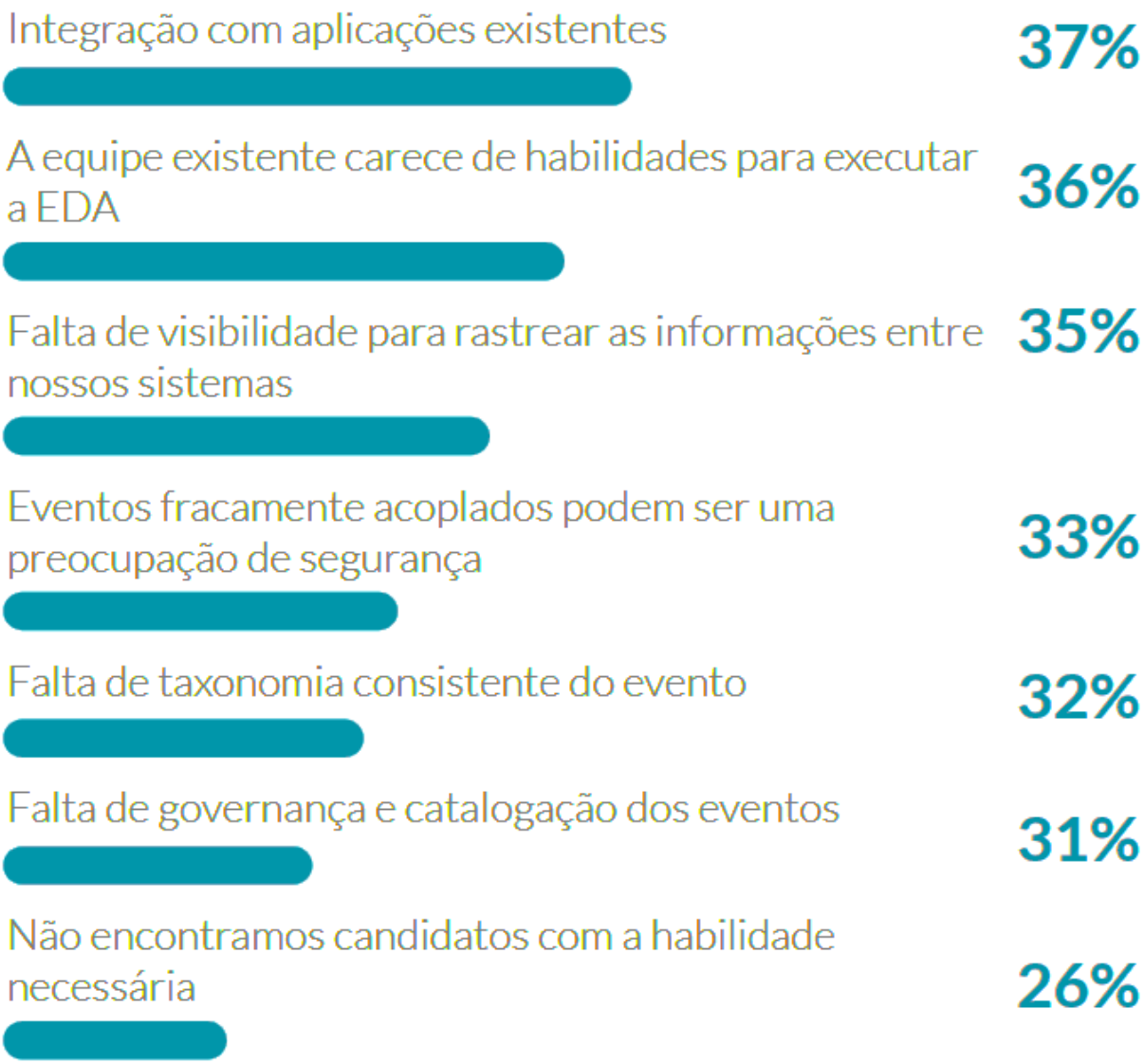

Figura 1.9: Resultado da pesquisa encomendada pela Solace – Desafios técnicos na implementação da EDA. Fonte: SOLACE, 2023.

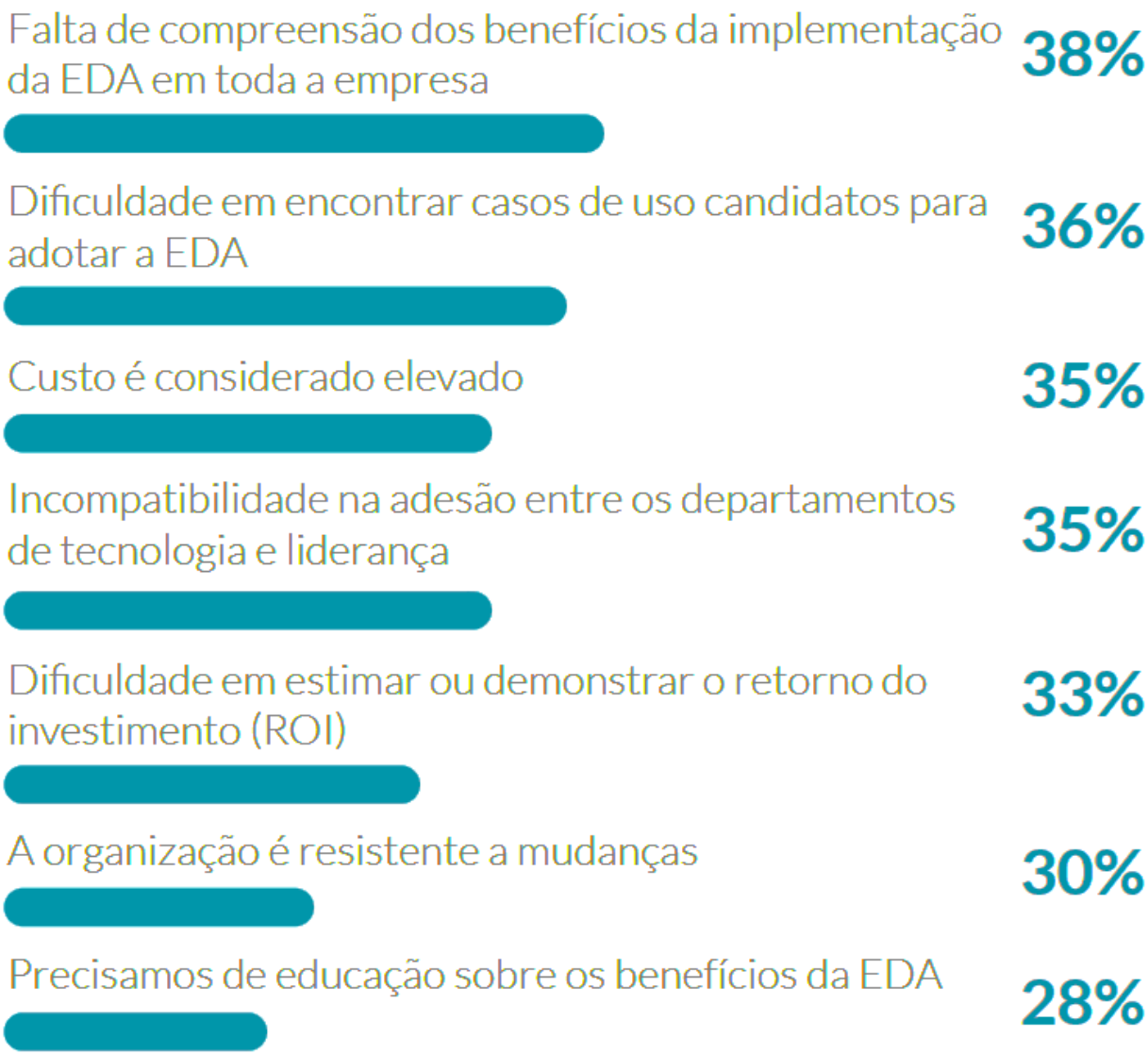

Figura 1.10: Resultado da pesquisa encomendada pela Solace – Desafios organizacionais na implementação da EDA. Fonte: SOLACE, 2023.

Concluindo

Neste capítulo, vimos o porquê da EDA, passando pela sua definição e a definição do que é evento. Vimos também os benefícios e os desafios que vêm junto ao adotá-la. Em seguida, entendemos o que é desenvolvimento de aplicações modernas e como ele se relaciona com a EDA. Por fim, vimos alguns dados de pesquisas e algumas previsões da adoção da EDA nas organizações. No próximo capítulo,

vamos entender os principais conceitos que cercam a Arquitetura orientada a eventos.

CAPÍTULO 2
Entendendo os principais conceitos

Neste capítulo vamos explorar os principais conceitos que cercam a EDA. Eles nos darão uma base de conhecimento para os capítulos seguintes. Mas antes, vamos entender o que é mensageria (*messaging*) e sua relação com EDA.

O que é mensageria?

Em uma definição interessante que retirei do site da Solace, em tradução livre, o termo **mensageria** refere-se à tecnologia "*que permite que sistemas de computador compartilhem informações sem exigir conexões diretas ou conhecimento da localização uns dos outros*" (https://docs.solace.com/Get-Started/what-is-messaging.htm).

Esse termo, também conhecido como *message-oriented middleware* (MOM) ou simplesmente *middleware*, existe bem antes de ouvirmos falar de EDA. Ele é utilizado quando nos referimos à **troca de mensagens entre sistemas de uma maneira mais ampla**, diferentemente da EDA, que é específica e segue os princípios de design e troca de informações por meio de eventos.

Muitos dos conceitos que vamos explorar a seguir servem para mensageria e EDA. Para facilitar, eles foram divididos em: *componentes-chaves*; *tipos de mensagem*; *anatomia de uma mensagem*; *formato da mensagem*; *protocolo de mensagem*; *destino da mensagem* e *garantia de entrega da mensagem*.

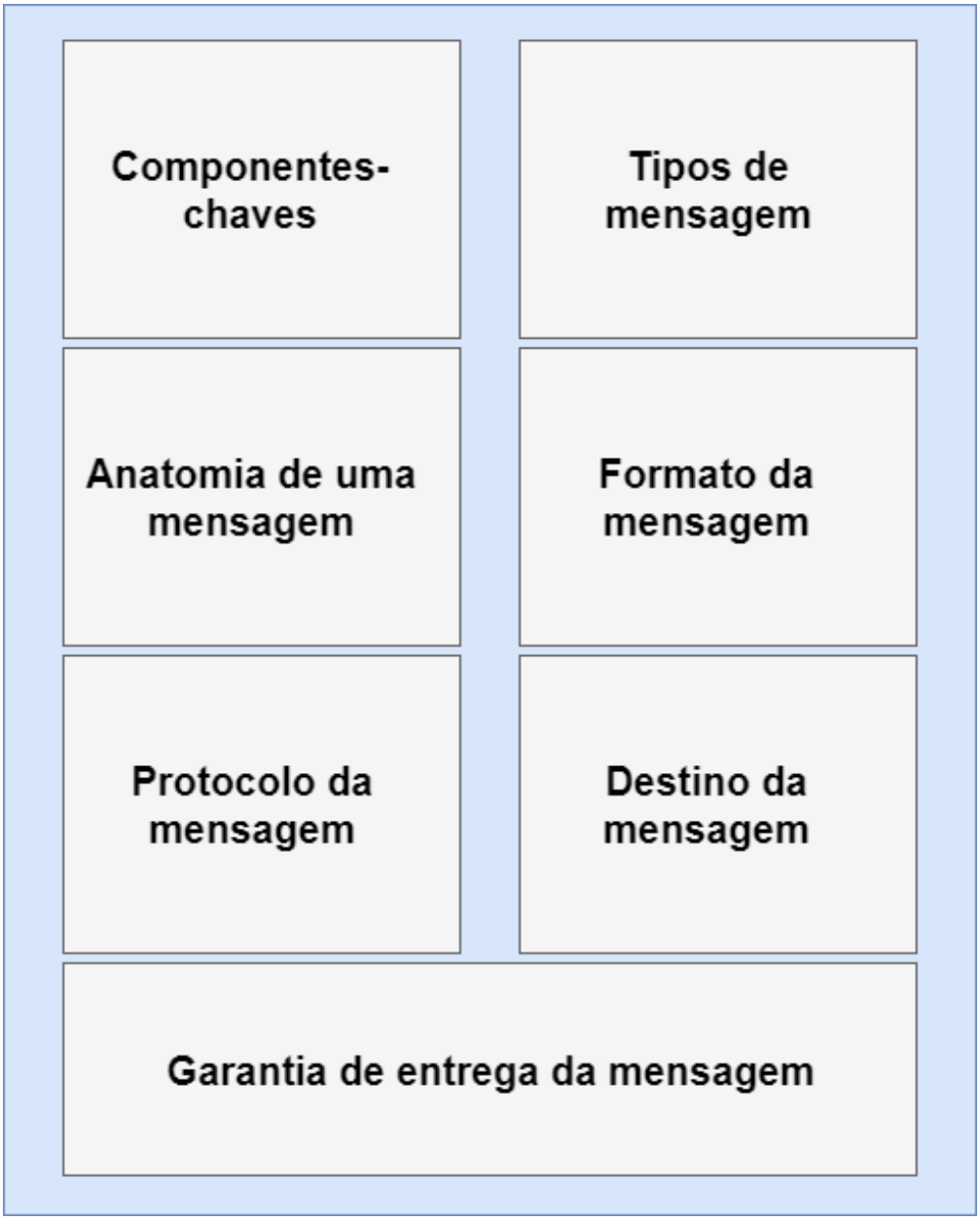

Figura 2.1: Visão geral dos principais conceitos que cercam a EDA.

2.1 Componentes-chaves

A EDA tem três componentes-chaves: o produtor (*event producer*) e o consumidor de eventos (*event consumer*) e, fazendo a ponte entre

eles, o *broker* de eventos (*event broker*).

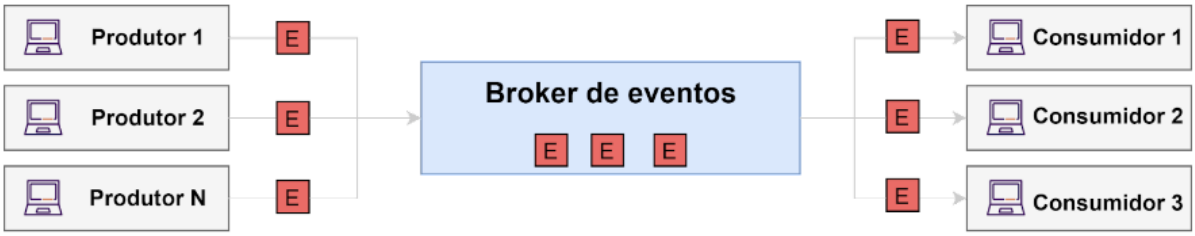

Figura 2.2: Modelo padrão da Arquitetura orientada a eventos.

Produtor e consumidor de eventos

Produtor e consumidor de eventos são aplicações que podem ser um programa, um processo, um microsserviço, um dispositivo da Internet das Coisas (IoT), um componente de integração ou qualquer outra aplicação executável que interage com o *broker* de eventos.

O produtor de evento publica um evento para o *broker* de eventos indicando que algo relevante ocorreu, e qualquer aplicação interessada (*consumidor*) pode usá-lo como gatilho para explorar alguma oportunidade, iniciar algum processamento, executar regras de negócio ou qualquer outro tipo de tarefa.

Uma aplicação pode fazer o papel somente de produtor ou consumidor de eventos ou pode fazer o papel de ambos. Por exemplo, um microsserviço de *pedido* publica um evento de *pedido criado* e o mesmo microsserviço consome um outro evento de *pagamento aprovado*, sendo esse o gatilho para atualizar a situação do pedido.

Um produtor de eventos também é conhecido como publicador de eventos (*publisher*) ou fonte de eventos (*event source*). No decorrer do livro, usaremos a expressão "produtor de eventos". Já um consumidor de eventos também é conhecido como subscritor de eventos (*subscriber*), coletor de eventos (*event sink*) ou manipulador

de eventos (*event handler*). No decorrer do livro, usaremos a expressão "consumidor de eventos".

Broker de eventos

O *broker* de eventos é uma solução/software que atua como uma ponte entre o produtor do evento e o consumidor do evento. Ele possui capacidades críticas que habilitam uma EDA distribuída, confiável e gerenciável com seu suporte ao modelo *publish-subscribe*.

O produtor de eventos publica o evento para o *broker* de eventos, que pode fazer o roteamento, tradução e persistência do evento e entregá-lo para os consumidores apropriados, desacoplando as aplicações produtoras e consumidoras de eventos.

O *broker* de eventos também é conhecido como *broker* de mensagem (*message broker*), barramento de eventos (*event bus*), roteador de eventos (*event router*) e *hub* de eventos (*events hub*). No decorrer do livro, usaremos o termo "*broker* de eventos". Na parte II, teremos capítulos dedicados aos *brokers* de eventos, explorando em detalhes suas capacidades críticas e seus tipos.

2.2 Tipos de mensagem

A forma como duas aplicações se comunicam trocando informações é através de mensagem, independente do tipo da comunicação utilizada, seja síncrona do modelo *request-response* ou assíncrona com EDA. Mensagem é um termo genérico que pode ser um evento (*event*), um *comando* (*command*) ou uma *query*.

Evento

Como vimos na definição de evento dada no capítulo anterior, um evento representa um fato significante, uma mudança de estado; ele

é imutável, uma vez que aconteceu, não pode ser desfeito. **Evento é um tipo de mensagem que segue os princípios de design da EDA**.

Podemos classificar o evento em duas categorias: *evento discreto* (*discrete event*) e *evento de série* (*event stream*).

Evento discreto não depende de outros eventos para fazer sentido. O evento discreto relata algo relevante que aconteceu, uma mudança de estado, como pedido criado, pagamento efetuado, preço atualizado, aluno entrou na academia etc.

Evento de série representa um fluxo contínuo de eventos, que são ordenados e medidos em um ponto do tempo, como medições do sensor de temperatura, métricas de recursos computacionais, ações do usuário em uma solução de análise de clique, sistema de monitoramento que recebe e processa registros de logs de diagnóstico em tempo real etc. Eventos de série são continuamente atualizados, representando o que aconteceu no passado e o que está acontecendo agora.

Comando

Comando é um tipo de mensagem que é uma solicitação a um sistema ou a uma outra aplicação para executar alguma ação ou alterar algum estado e às vezes espera uma resposta.

Por exemplo, uma rotina de processar pagamento executa uma ação indicando se o pagamento foi bem-sucedido ou não. Já um exemplo de alteração de estado: uma rotina de alteração de preço executa uma ação para alterar o preço do produto, mudando consequentemente o seu estado. Utilizando uma implementação REST API, seriam os métodos *POST*, *PUT* e *DELETE*.

Os comandos são executados de forma síncrona e normalmente indicam uma conclusão. Porém, conseguimos utilizar comandos de forma assíncrona, tipicamente seguindo o modelo de troca de

mensagem *point-to-point* ou *request/reply*, que veremos em detalhes nos capítulos seguintes.

Query

Query é um tipo de mensagem que é uma solicitação a um sistema ou a uma outra aplicação para recuperar informações. Diferentemente do comando, cuja resposta é opcional, uma *query* requer uma resposta.

Por exemplo, uma rotina para listar as informações de produto, onde as informações do produto são retornadas na resposta de uma solicitação. Utilizando uma implementação REST API, seriam os métodos *GET* e *HEAD*.

As *queries* são executadas de forma síncrona e requerem uma resposta. Também conseguimos utilizar *queries* de forma assíncrona, tipicamente seguindo o modelo de troca de mensagem *request-reply*, que veremos em detalhes nos capítulos seguintes.

2.3 Anatomia de uma mensagem

A mensagem é o meio de comunicação entre as aplicações e sistemas. Não importa o tipo de mensagem, se é um *evento*, um *comando* ou uma *query*, ela tem a mesma estrutura. A mensagem tem duas partes principais: o cabeçalho (*header*) e o corpo da mensagem (também conhecido como *payload* ou *body*).

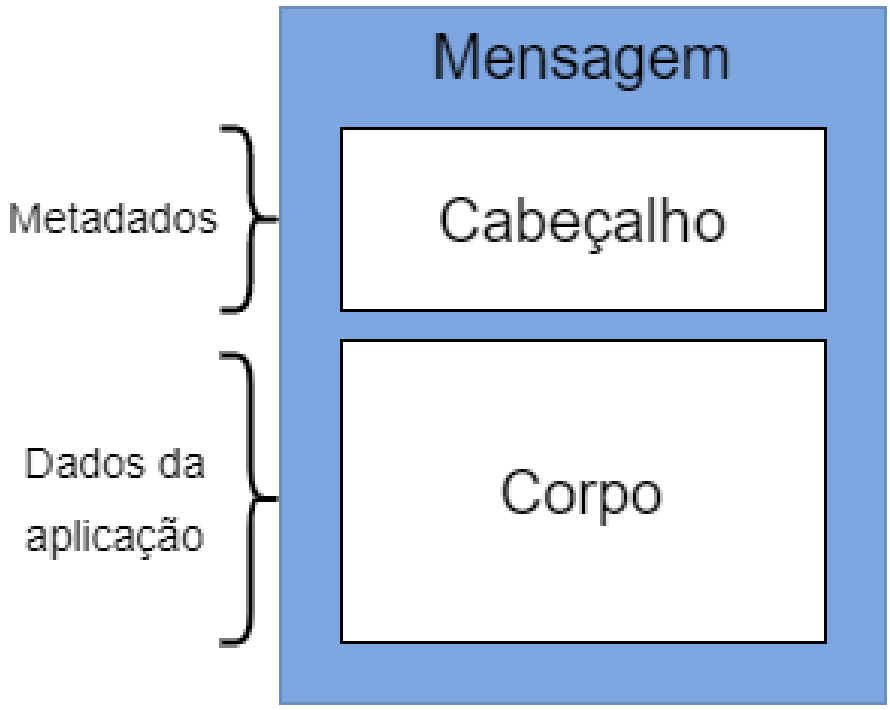

Figura 2.3: Anatomia da mensagem.

Cabeçalho

O cabeçalho (*header*) carrega os metadados da mensagem, que são os atributos da mensagem. Esses atributos podem conter a identificação da aplicação que publicou o evento, a data e hora em que o evento chegou, o tipo da mensagem etc.

Os *brokers* de eventos utilizam esses atributos para definir/executar algum tipo de comportamento, como fazer o roteamento da mensagem, rastreamento das mensagens, entre outros.

Esses atributos também podem carregar informações específicas das aplicações. Por exemplo, o ID do pedido que um sistema usa como um identificador exclusivo para rastrear todos os eventos relacionados a uma determinada compra. Esses atributos específicos das aplicações também são conhecidos como **propriedades**. Algumas APIs definem suas próprias propriedades. Por exemplo, o JMS definiu propriedades como *JMSXUserID* e *JMSXAppID*, que

podem ser usadas para identificar o usuário ou aplicação que envia a mensagem.

Corpo

O corpo da mensagem contém as informações que estão sendo transmitidas entre as aplicações, como informações do pedido, informações do aluno que entrou na academia, dados capturados e transmitidos por sensores etc.

O produtor da mensagem envia essas informações em um formato específico para o *broker* de eventos, podendo ser em dados binários ou dados estruturados, como texto, o qual é interpretado e entendido pelo consumidor da mensagem. Ambos devem concordar com o formato do corpo da mensagem para que as informações possam ser interpretadas adequadamente.

2.4 Formato da mensagem

A mensagem utilizada na comunicação entre aplicações através do *broker* de eventos precisa de um formato (*data format*) e um esquema (*schema*) definido.

Imagina se você se depara com esta informação: `{Maria,852658996345,azul}`. Você consegue interpretá-la? Maria parece ser um nome, certo? E o número, o que significa? E o `azul`, é a cor dos olhos da Maria? Sem um formato e um esquema definidos, nem nós, humanos, nem a máquina vai conseguir interpretar a informação de maneira adequada.

Um formato é um conjunto de nomes e tipos de campos que caracterizam a estrutura de uma mensagem. Um formato baseado em texto popular é o `JSON`, que é autodescritivo, o que significa que pode ser analisado sem fazer referência a um esquema.

```
{
    "nome": "Maria",
    "cpf": "85265899635",
    "cor_preferida": "azul"
}
```

No nosso exemplo do formato `JSON`, apesar de sabermos que o número `85265899635` corresponde ao CPF da Maria, como podemos validar se o CPF está em um formato válido? Como podemos garantir que os campos obrigatórios serão informados? Como podemos validar se a cor preferida informada tem um nome válido de cor? Como podemos garantir que o nome informado (no nosso exemplo, Maria) respeitará o limite de caracteres definido?

Apesar de o formato `JSON` ser autodescritivo, podemos definir um esquema para ele, o *JSON esquema*. Com um esquema definido, tanto a aplicação produtora quanto a aplicação consumidora do evento podem garantir que o conteúdo do evento seja devidamente validado, entendido e interpretado, seguindo as regras definidas no esquema.

Um esquema define a estrutura e o significado de cada valor dos campos no corpo da mensagem para que o produtor e todos os consumidores entendam a estrutura e o que cada valor significa.

Ao definir o formato do corpo da mensagem, primeiro devemos escolher se desejamos usar um formato baseado em texto (*text-based format*) ou binário (*binary format*).

Formato baseado em texto

No formato baseado em texto, o corpo da mensagem é em texto simples. Esse formato é adequado para dados que precisam ser visualizados por pessoas, como arquivamento de dados que precisam ser auditáveis, análise de log de aplicações de modo geral para investigar problemas e assim por diante.

Como vimos, o `JSON` é um formato baseado em texto, mas existem outros, como `XML`, `CSV` (Valores Separados por Vírgula).

O problema da mensagem no formato baseado em texto é que ela tende a ser muito grande, pois tanto o conteúdo em si quanto sua definição de estrutura (nome dos campos, como *nome*, *cpf* e *cor_preferida* do nosso exemplo) fazem parte do corpo da mensagem, além de cada dígito ser convertido no formato ASCII, que ocupa mais espaço em comparação ao formato binário. Dependendo do tamanho da mensagem, isso influencia consideravelmente no desempenho da comunicação entre aplicações através da rede e no seu armazenamento.

Utilizando web APIs no modelo *request-response*, o formato baseado em texto tem dominado por anos, primeiro com o uso de `XML` e depois com o uso do `JSON`. Esses formatos são mais simples de usar e amplamente conhecidos e suportados por qualquer linguagem de programação, bem como ferramentas de integração e plataformas de desenvolvimento diversas.

A Gartner recomenda que você avalie primeiro se pode atender aos seus requisitos de desempenho e largura de banda de rede usando `JSON` ou `XML`. Isso lhe dará a mais alta usabilidade e amplo suporte. `JSON` ou `XML` também fornecerá uma melhor experiência do desenvolvedor em EDA e no modelo de *request-response*.

Formato binário

O formato binário é mais compacto do que o formato baseado em texto, pois sua definição de estrutura não vai no corpo da mensagem. Formatos binários requerem bibliotecas de serialização/desserialização (*serialization/deserialization*) e o corpo da mensagem não pode ser lido a menos que seja decodificado (https://pt.wikipedia.org/wiki/Serializa%C3%A7%C3%A3o).

Os formatos binários mais comuns são o *Apache Avro* e o *Protocol Buffer* (*ProtoBuf*), que são padrões de mercados independentes de linguagem de programação e/ou plataforma.

Apache Avro é um sistema de serialização de dados que provê uma rica estrutura de dados. É um formato binário compacto e eficiente (https://avro.apache.org/).

ProtoBuf é um mecanismo de serialização de dados criado pelo Google no início dos anos 2000 e disponibilizado para a comunidade em 2008. Seu principal objetivo foi fornecer uma maneira mais eficiente na comunicação entre sistemas em comparação ao `XML`, que era o padrão baseado em texto mais utilizado na época (https://developers.google.com/protocol-buffers/).

Diferentemente do formato baseado em texto, o formato binário tem seu conteúdo em bytes (0s e 1s), que não é legível por pessoas. Para conseguir interpretá-lo, é necessária a desserialização do seu conteúdo.

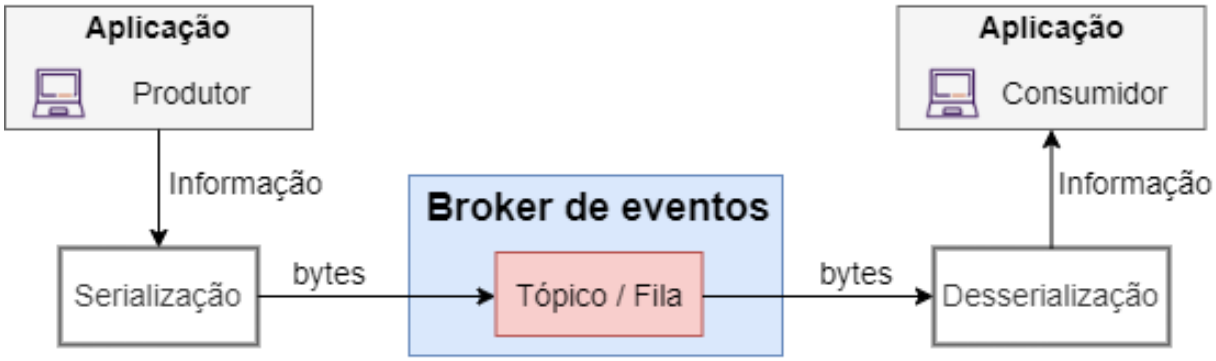

Figura 2.4: Exemplo de envio de dados serializados para um broker de eventos.

O ponto principal é se o formato escolhido é padrão de mercado, independente de linguagem de programação e plataforma, e se possui um amplo suporte de biblioteca para serialização e desserialização.

Uma estratégia normalmente adotada é: se precisamos expor nossos dados através de APIs para nossos clientes e parceiros externos, dê

preferência aos formatos baseados em texto, como o `JSON`, devido à sua simplicidade e ao amplo conhecimento e uso no mercado. Já para os dados utilizados na comunicação interna entre componentes/microsserviços, poderíamos adotar o formato binário, por ser mais compacto e com melhor desempenho na comunicação através da rede, caso o armazenamento e a largura de banda de rede seja uma preocupação.

Na seção *Gerenciamento de esquema* do capítulo 5, *Broker de eventos habilita a arquitetura orientada a eventos*, exploraremos o **gerenciamento de esquema** (*schema management*), onde abordaremos temas importantes relacionados a esquema, como registro de esquema (*schema registry*), evolução ou versionamento dos esquemas (*schema evolution or versioning*), modos de compatibilidade – como compatibilidade com versão anterior (*backward compatibility*) e compatibilidade com versão futura (*forward compatibility*) –, entre outros.

2.5 Protocolo de mensagem

Quando falamos em protocolo, estamos falando de protocolo de comunicação, que é a forma com que as aplicações e/ou sistemas se comunicam seguindo um conjunto de regras (https://simple.wikipedia.org/wiki/Communication_protocol). Dentro do nosso contexto de EDA e mensageria, vamos seguir com o termo *protocolo de mensagem* (*messaging protocol*).

Os **protocolos de mensagem** ditam as regras de como as aplicações se conectam aos *brokers* de eventos para publicar e/ou consumir eventos. Eles determinam como será o funcionamento interno dos *brokers* de eventos, bem como sua arquitetura, e influenciam nos objetivos dos clientes em relação à largura de banda e à qualidade do serviço (*quality of service*, ou QoS), como garantia

de entrega de mensagens, que veremos em breve ainda neste capítulo.

Os *brokers* de eventos normalmente suportam uma variedade de protocolos de mensagens, termo conhecido como *multiprotocolo* (*multi-protocol*). Os protocolos de mensagens mais utilizados são os de **padrão aberto** – HTTP/S; *Advanced Message Queuing Protocol* (AMQP); *Message Queuing Telemetry Transport* (MQTT) – e o **protocolo proprietário** Apache Kafka (*wire protocol*).

Quando nos referimos ao protocolo HTTP/S, estamos falando em REST API. Muitos *brokers* de eventos contemplam esse tipo de protocolo. Além de permitir que aplicações publiquem e consumam eventos, esse tipo de protocolo é utilizado para tarefas administrativas, como gerenciamento do *broker*, criação e remoção de filas e tópicos e por aí vai.

O protocolo AMQP, *Advanced Message Queuing Protocol* (https://www.amqp.org/), é um protocolo leve para troca de mensagens entre aplicações mantido pela OASIS (https://www.oasis-open.org/). Uma característica importante desse protocolo é seu modelo de roteamento de mensagem/eventos (https://www.rabbitmq.com/tutorials/amqp-concepts.html). As mensagens são publicadas em *exchanges*, que são roteadas para diferentes filas (*queues*) usando regras chamadas de *bindings*. Os consumidores buscam mensagens dessas filas.

O protocolo *AMQP* é implementado por diversos *brokers* de eventos. Os mais conhecidos deles são o RabbitMQ (https://www.rabbitmq.com/), o Apache ActiveMQ (https://activemq.apache.org/) e o Amazon MQ (https://aws.amazon.com/pt/amazon-mq), que é um *broker* de evento/mensagem gerenciado da AWS que roda com RabbitMQ ou Apache ActiveMQ. Esses *brokers* de eventos têm suas particularidades e seus diferenciais, mas seguem o mesmo conjunto de regras deste protocolo.

O protocolo *Message Queuing Telemetry Transport* (MQTT) é um protocolo simples e leve de padrão aberto para troca de mensagens entre máquinas (M2M). Ele foi criado pela IBM no final dos anos 90 com o objetivo de conectar sensores em oleodutos a satélites. Assim como o AMQP, o MQTT também é mantido pela OASIS (https://mqtt.org/). Ele é leve e ideal de ser utilizado em dispositivos de hardware com uma alta limitação de largura de banda de rede e com alta latência, fazendo dele o protocolo padrão para Internet das Coisas (IoT).

O protocolo MQTT é suportado por *brokers* de eventos dedicados, como o Eclipse Mosquitto (https://mosquitto.org/) e o HiveMQ (https://www.hivemq.com/). Outros *brokers* de eventos, que são multiprotocolo, também têm algum tipo de suporte ao protocolo MQTT, como o RabbitMQ, Solace e ActiveMQ. Existem também as plataformas de Internet das Coisas (IoT) que vão além do protocolo, tais como Microsoft Azure IoT Hub (https://azure.microsoft.com/pt-br/products/iot-hub/) e AWS IoT (https://aws.amazon.com/pt/iot/).

Nada impede que um *broker* de eventos defina seu próprio protocolo de comunicação, o qual chamamos de **protocolo proprietário**. O Apache Kafka é um bom exemplo. Seus criadores desenvolveram seu próprio protocolo. Todos os produtos e serviços que são desenvolvidos tendo o Apache Kafka como base utilizam esse protocolo proprietário, como o Confluent (https://www.confluent.io/), o AWS MSK (https://aws.amazon.com/pt/msk/) e o Event Hubs, da Azure (https://azure.microsoft.com/pt-br/products/event-hubs).

Mas por que um *broker* de eventos não utiliza um protocolo aberto já consolidado e amplamente utilizado e testado no mercado, como o AMQP, MQTT ou até mesmo o HTTP/S? Por que ele reinventa a roda e cria seu protocolo? No próprio site oficial do Apache Kafka (https://kafka.apache.org/protocol.html) tem um tópico bem interessante: *Algumas perguntas filosóficas comuns* (*Some Common Philosophical Questions*). Essa referência está em inglês, mas lá estão as respostas para essas perguntas. Vale a leitura!

A seguir, em tradução livre, vou extrair parte das respostas de três perguntas desse tópico do site oficial do Apache Kafka. Além disso, vou tentar resumir em poucas palavras o motivo por trás de cada resposta.

Por que o Apache Kafka não usa HTTP?

"*O protocolo proprietário permite que nossos clientes façam uso de alguns dos recursos TCP mais avançados*" — ele é um protocolo binário sobre TCP (KAFKA, c2023; tradução livre).

Motivo: Utilização de recursos mais avançados que o HTTP não suporta.

E se o Apache Kafka suportasse outros protocolos?

"*Pela nossa experiência, isso torna muito difícil adicionar e testar novas funcionalidades se elas tiverem que ser compatíveis com muitas implementações de protocolo. Nossa sensação é de que a maioria dos usuários realmente não vê vários protocolos como uma funcionalidade, eles só querem um boa aplicação cliente confiável na linguagem de programação de sua escolha*" (KAFKA, c2023; tradução livre).

Motivo: Velocidade e agilidade para disponibilizar novas funcionalidades, alinhado à sensação de que os usuários não se importam em ter vários protocolos.

Por último, por que o Apache Kafka não adota o AMQP ou algum outro protocolo?

"*No geral, o problema é que o protocolo determina grandes partes da implementação e não poderíamos fazer o que fazemos se não tivéssemos controle sobre o protocolo. Acreditamos que é possível fazer melhor (...) para fazer isso precisamos construir algo que funcione de forma diferente*" (KAFKA, c2023; tradução livre).

Motivo: Flexibilidade, adaptação e controle no protocolo.

Um outro protocolo que vale a pena comentar é o protocolo **Websocket**. Ele é um protocolo com comunicação bidirecional projetado para comunicação em tempo real com conexões *point-to-point* TCP/TLS e não tem suporte nativo ao modelo *publish-subscribe*. Podemos combinar o protocolo Websocket com os protocolos MQTT, AMQP ou qualquer outro protocolo proprietário para fornecer comunicações no modelo *publish-subscribe*. O uso desses protocolos de mensagens fornece uma comunicação mais estruturada e resiliente do que um Websocket nativo.

2.6 Destino da mensagem

Quando uma aplicação produtora publica eventos para o *broker* de eventos indicando que algo relevante ocorreu, de alguma forma o evento é endereçado a um **destino** (*destination*), também conhecido como **canal** (*channel*). Esse destino pode ser uma fila (*queue*) ou um tópico (*topic*). A diferença está basicamente em como o evento será distribuído/entregue para as aplicações consumidoras.

Fila

Se o destino for do tipo *fila*, isso significa que o evento pode ser **consumido por no máximo uma aplicação consumidora**. Se não houver aplicações consumidoras disponíveis no momento em que o evento foi enviado, ele poderá ser mantido na fila até que esteja disponível uma aplicação consumidora que possa processar o evento. Se uma aplicação consumidora receber o evento e não confirmar antes de encerrar a conexão com o *broker* de eventos, o evento será reenviado para outra aplicação consumidora. Uma fila pode ter mais de uma aplicação consumidora *escutando* ela, porém os eventos serão balanceados entre essas aplicações consumidoras disponíveis.

Tópico

Se o destino for do tipo *tópico*, isso significa que o evento pode ser **consumido por mais de uma aplicação consumidora**. Quando publicamos um evento em um destino do tipo tópico, esse evento será entregue para todas as aplicações consumidoras interessadas/assinadas ao tópico — de zero a muitas aplicações consumidoras que assinaram o tópico receberão uma cópia do evento. Apenas as que tiverem uma assinatura ativa no tópico receberão uma cópia do evento.

Em alguns *brokers* de eventos, as nomenclaturas podem variar. Por exemplo, para *brokers* de eventos com o protocolo AMQP, as *exchanges* são consideradas um tipo de tópico. Veremos no capítulo seguinte os modelos de troca de mensagens *publish-subscribe* e *point-to-point* em detalhes. Esses modelos têm uma relação direta com o tipo de destino do evento/mensagem.

2.7 Garantia de entrega da mensagem

Como sabemos, o produtor publica eventos para o *broker* de eventos, que faz a entrega dos eventos para os consumidores apropriados, desacoplando as aplicações produtoras e consumidoras.

Por causa da natureza desacoplada e assíncrona da EDA, precisamos entender qual garantia o *broker* de eventos oferece em relação à entrega dos eventos para os consumidores, para que, com isso, tanto as aplicações produtoras quanto as aplicações consumidoras possam ser desenvolvidas de acordo. Essas aplicações e/ou o próprio *broker* de eventos podem falhar e ficar indisponíveis ao publicar e consumir eventos.

A garantia de entrega (*guaranteed delivery*) é uma capacidade do *broker* de garantir que o evento seja entregue para as aplicações

consumidoras, mesmo em circunstâncias de falhas ou indisponibilidade das aplicações ou do próprio *broker*. A garantia de entrega está diretamente relacionada ao nível de qualidade do serviço (QoS) que um *broker* de eventos pode oferecer.

Uma das capacidades que um *broker* de eventos tem é a confirmação do recebimento de eventos (*Acknowledgement*, ou simplesmente *Ack*). Essa confirmação pode ocorrer em dois momentos no fluxo de publicação e consumo de eventos:

1. O *broker* de eventos confirma à aplicação produtora que o evento foi recebido;
2. A aplicação consumidora confirma ao *broker* que o evento foi processado (ou pelo menos recebido).

Existem alguns modos de confirmação no contexto da aplicação consumidora, como a confirmação automática ou manual. Esses tipos de confirmação serão vistos em detalhes no capítulo 6, *Funcionalidades comuns de um broker no fluxo de publicação e consumo de eventos*.

Uma outra característica comum dos *brokers* de eventos é a sua capacidade de persistir/armazenar mensagens ou eventos. As mensagens são persistidas (*persistent messaging*) no banco de dados interno do *broker* de eventos para garantir que não sejam perdidas em caso de falhas ou indisponibilidades.

Uma vez que as mensagens ou eventos podem ser persistidos no banco de dados, essas mensagens podem ficar disponíveis para as aplicações consumidoras que ainda não estão conectadas ou que perderam a conexão com o *broker* de eventos.

Duas características comuns de consumidores que normalmente geram confusão com mensagem persistente são os consumidores duráveis (*Durable Consumers* ou *Durable Subscribers*) e não duráveis (*Nondurable Consumers* ou *Nondurable Subscribers*).

Consumidores duráveis: são consumidores que, mesmo off-line (não conectados com o *broker*), poderão receber as mensagens quando ficarem conectados com o *broker* de eventos.

Consumidores não duráveis: são consumidores que vão receber as mensagens somente se estiverem conectados ao *broker* de eventos no momento em que a mensagem foi recebida pelo *broker*.

Só é possível ter consumidores duráveis por causa da persistência da mensagem no banco de dados, considerada uma memória não volátil.

Imagine as seguintes situações possíveis de falhas que podem ocorrer nesse fluxo de publicação e consumo de eventos:

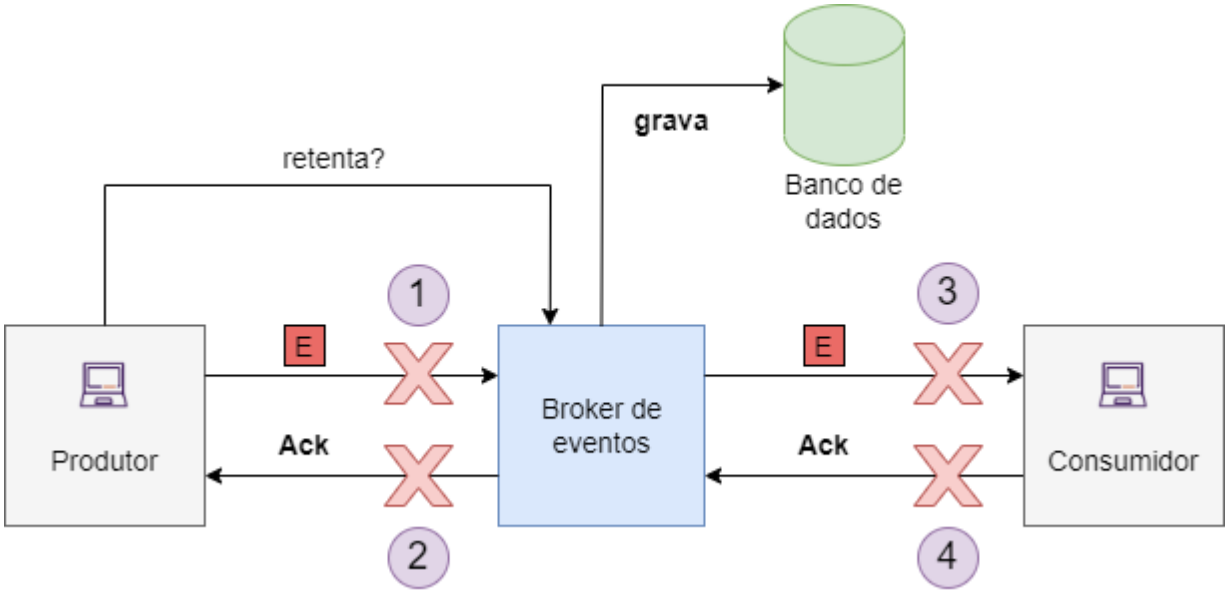

Figura 2.5: Possíveis situações de falha no fluxo de publicação e consumo de eventos.

1. A aplicação produtora perde a conectividade com o *broker*;
2. O *broker* recebe e grava o evento no seu banco de dados interno, porém, antes de confirmar o recebimento, a conexão

com a aplicação produtora é perdida ou alguma falha no *broker* ocorre;
3. A aplicação consumidora perde a conectividade com o *broker*;
4. A aplicação consumidora recebe o evento, porém, antes de confirmar o recebimento, a conexão é perdida com o *broker* ou alguma falha inesperada ocorre na aplicação.

Nessas possíveis situações de falha ou indisponibilidade, como as aplicações produtoras e consumidoras e o *broker* deveriam se comportar? Precisamos entender quais semânticas de entrega (*Delivery Semantics*) o *broker* de evento suporta. Essas semânticas ditam o nível de serviço oferecido pelo *broker* de eventos.

A seguir, vamos explorar três semânticas de entrega: entrega no máximo uma vez (*At-most-once delivery*), entrega pelo menos uma vez (*At-least-once delivery*) e entrega exatamente uma vez (*Exactly-once delivery*).

Entrega no máximo uma vez

A semântica de *entrega no máximo uma vez* é considerada a mais simples de ser implementada. Nessa semântica, **é tolerável a perda de eventos ou mensagens, porém ela garante que a mensagem NÃO será duplicada**.

Quando a aplicação produtora publica o evento e recebe um erro interno do *broker* de eventos, a aplicação produtora não republica o evento, pois nada garante que o *broker* não vai entregar o evento para o consumidor.

Como a perda da mensagem é tolerável, a mensagem não precisa necessariamente ser persistida nem confirmada pelo *broker*; com isso, otimiza-se o desempenho do fluxo de publicação e consumo.

Para casos de usos onde **mensagem duplicada é um problema e a perda da mensagem é tolerada**, um *broker* de eventos que suporte essa semântica no máximo uma vez é a escolha mais apropriada.

Um exemplo de uma possível solução com essa semântica é o consumidor do evento ser uma aplicação de envio de e-mail. Não queremos ficar enviando *spam* para nosso cliente, mas garantir que ele receberá no máximo um e-mail — se por acaso ele não receber o e-mail, não será um problema para o negócio.

Entrega pelo menos uma vez

A semântica de *entrega pelo menos uma vez* é a mais comumente suportada pelos *brokers* de eventos. Nessa semântica, **a perda de eventos ou mensagens NÃO é tolerada**, e **mensagens duplicadas podem ocorrer** em cenários de falha ou indisponibilidade.

Quando a aplicação produtora publica o evento e recebe um erro do *broker* de eventos e/ou não recebe a confirmação de recebimento, a aplicação produtora republica o evento, de acordo com sua configuração de retentativas. Como não é garantido que o *broker* processou o evento anterior, o evento reenviado pode ser processado novamente pela aplicação consumidora.

Como a perda da mensagem não é tolerável, a mensagem precisa ser persistida e confirmada pelo *broker*. Isso pode impactar no desempenho do fluxo de publicação e consumo em comparação à semântica de *entrega no máximo uma vez*.

Para casos de uso onde a **perda da mensagem NÃO é tolerada**, um *broker* de eventos que suporte a semântica *pelo menos uma vez* é a escolha mais apropriada.

Um exemplo de uma possível solução que usaria essa semântica é o caso de o consumidor do evento ser uma aplicação que notifica o hospital mais próximo de que um acidente ocorreu. Não é um problema notificar o hospital duas ou mais vezes, porém seria um grande problema não o notificar.

Entrega exatamente uma vez

A semântica de *entrega exatamente uma vez* é a mais difícil de ser implementada e poucos *brokers* de eventos a suportam. Nessa semântica, **a perda e duplicação de eventos ou mensagens NÃO é tolerada**. Ela garante que a aplicação consumidora receberá um e somente um evento, sem eventos duplicados. Para alcançar essa semântica, é necessária a cooperação das aplicações produtoras e consumidoras e do próprio *broker* de eventos.

Quando a aplicação produtora publica o evento e recebe um erro do *broker* de eventos e/ou não recebe a confirmação de recebimento, a aplicação produtora republica o evento — mesmo processo de envio da semântica *entrega pelo menos uma vez*. Porém, a aplicação produtora obrigatoriamente envia um identificador único em cada evento. Esse identificador não é alterado em caso de retentativa por parte da aplicação produtora.

Essa chave enviada pela aplicação produtora é validada pelo *broker* de eventos antes de gravar o evento na sua base de dados interna. Essa validação descarta qualquer evento com o mesmo identificador já recebido. Para alcançar esse comportamento, a aplicação produtora e o *broker* de eventos precisam trabalhar em conjunto.

No contexto da aplicação consumidora, caso perca a conectividade com o *broker* de eventos ou ocorra algum erro interno, ela precisa controlar se o evento já foi processado ou não — em um cenário de erro, nada garante que a confirmação de recebimento ocorreu, e o mesmo evento pode ficar disponível novamente para processamento.

Na verdade, essa semântica é o que todos querem: sem perda de eventos nem eventos duplicados. Mas nem todo *broker* de eventos nem todo protocolo suporta essa semântica. A validação extra do *broker* de eventos para descartar eventos duplicados pode sobrecarregar o desempenho do fluxo de publicação e o consumo de eventos como um todo. Nesse caso, será necessário avaliar esse impacto na solução.

Um exemplo de um caso de uso em que a perda e a duplicidade da mensagem são um problema é uma solução de e-commerce. Não queremos perder um pedido e também não queremos enviar o mesmo pedido mais de uma vez para o cliente.

Uma alternativa à semântica de entrega exatamente uma vez

Uma vez que nem todos os *brokers* de eventos suportam essa semântica por causa da sua complexidade – pois envolve a cooperação de todos os componentes envolvidos no fluxo de publicação e consumo de eventos –, como alternativa para garantir o não processamento de eventos duplicados, podemos usar a semântica de *entrega pelo menos uma vez* seguindo o princípio de design na aplicação consumidora chamada ***consumidor idempotente*** (*idempotent consumer*). Aplicações desenvolvidas com esse princípio fazem com que a solução como um todo seja mais resiliente e flexível.

> **CONSUMIDOR IDEMPOTENTE**
>
> O resultado de processar o mesmo evento várias vezes deve ser o mesmo que processar o evento uma vez.
>
> Para desconsiderar um evento duplicado, a aplicação consumidora deve ter a capacidade de fazer esse controle, o que normalmente é realizado armazenando o ID do evento em uma base de dados de controle para verificar se o evento com um determinado ID já foi processado. O *broker* de eventos garante o ID único de cada evento. Pode ser configurada uma janela nesse controle — por exemplo, o mesmo evento que chegar depois de 10 minutos não é mais considerado um evento duplicado.

Na semântica de *entrega exatamente uma vez*, esse princípio de design também se aplica às aplicações produtoras e consumidoras. As aplicações produtoras precisam garantir a unicidade dos eventos

enviados para que o *broker* de eventos possa fazer esse descarte dos duplicados.

Concluindo

Neste capítulo, entendemos o que é o produtor e o consumidor dos eventos e o *broker* de eventos, que são considerados os componentes essenciais. Exploramos os principais conceitos que cercam essa arquitetura, como os tipos, a anatomia, os formatos, os protocolos e o destino das mensagens. Por fim, vimos a importância dos conceitos de qualidade do serviço, relacionados à garantia de entrega das mensagens. No próximo capítulo, vamos explorar os padrões arquiteturais relacionados à EDA.

CAPÍTULO 3
Padrões de arquitetura para a Arquitetura orientada a eventos

O objetivo deste capítulo é entendermos os fundamentos de alguns padrões de arquitetura e como podemos utilizá-los dentro da nossa solução de EDA.

O que é padrão de arquitetura?

Um padrão de arquitetura é uma solução reutilizável para um problema que ocorre com frequência dentro de um determinado contexto. Em resumo, é uma solução recorrente para um problema recorrente (https://pt.wikipedia.org/wiki/Padr%C3%A3o_de_arquitetura).

Apesar de os padrões que vamos explorar serem comprovadamente utilizados e bem-sucedidos em diversos projetos de EDA, precisamos selecioná-los de acordo com nosso caso de uso e aplicá-los levando em consideração as capacidades e características do *broker* de eventos que adotaremos.

Vale destacar que vamos explorar os fundamentos dos padrões mais comuns e mais utilizados relacionados à EDA, mas que existem outros. Na *Bibliografia*, ao final do livro, há referências de leitura dos padrões que vamos explorar e de muitos outros, mas que estão fora do escopo deste livro.

Os padrões que veremos foram agrupados em:

- **Padrões de entrega de evento**;
- **Padrões de comunicação de mudança de estado**;
- **Outros padrões**.

Figura 3.1: Visão geral dos padrões de arquitetura.

3.1 Padrões de entrega de evento

Os padrões de entrega de evento referem-se à comunicação assíncrona através de mensagens ou eventos (*Asynchronous Message*) entre os produtores e consumidores por meio dos canais, seja por fila ou tópico. Essa comunicação desacoplada é onde a EDA brilha por intermédio de um *broker* de eventos.

Podemos ter variações na quantidade de produtores publicando e na quantidade de consumidores consumindo o evento. Elas são: um-para-um (*one-to-one*); muitos-para-muitos (*many-to-many*); um-para-muitos (*one-to-many*); e muitos-para-um (*many-to-one*).

Point-to-point

A comunicação no padrão *point-to-point* **refere-se a quando eventos publicados por um único produtor são consumidos por um e somente um consumidor** por meio de um canal do tipo fila. Esse padrão também é conhecido como *um-para-um*.

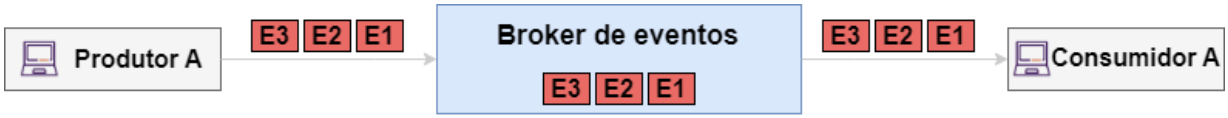

Figura 3.2: Entrega de evento point-to-point.

Uma variação desse padrão é o compartilhamento do mesmo canal com mais de um consumidor, também conhecido como *um-para-muitos*. O objetivo é escalar a aplicação consumidora para processar os eventos em paralelo e ter uma maior tolerância a falhas, uma vez que a aplicação consumidora poderia estar em máquinas distintas. Além disso, essa variação é também conhecida como *consumidores concorrentes* (*competing consumers*), *grupo de consumidores* (*consumer groups*) e *filas de trabalho* (*work queues*), e é normalmente usada por diversas instâncias da mesma aplicação/microsserviço dentro do mesmo grupo.

Figura 3.3: Entrega de evento point-to-point com mais de um consumidor.

Não importa quantos consumidores estejam compartilhando o mesmo canal, **apenas um consumidor vai receber um evento em particular**.

Um exemplo de uma possível solução que usaria esse padrão é o consumidor do evento ser uma aplicação de processamento de imagem, pois haveria várias instâncias (possivelmente em máquinas distintas) executando em paralelo para processar imagens o mais rápido possível.

Publish/subscribe

A comunicação no padrão *publish-subscribe* (ou simplesmente *pub/sub*) **refere-se a quando eventos publicados pelos produtores são distribuídos para diferentes consumidores interessados**. Cada consumidor registrado no canal do tipo tópico recebe sua própria cópia do evento. É também conhecido como *um-para-muitos*, *fan-out* e *broadcast*.

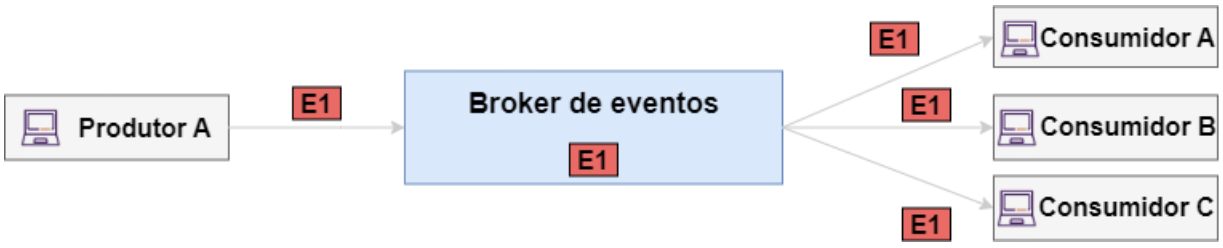

Figura 3.4: Entrega de evento pub/sub.

Esse padrão não se limita a um único produtor e muitos consumidores (*um-para-muitos*). Conseguimos utilizá-lo com outras variações, como *um-para-um* (com o padrão *point-to-point*), *muitos-para-um* e *muitos-para-muitos*.

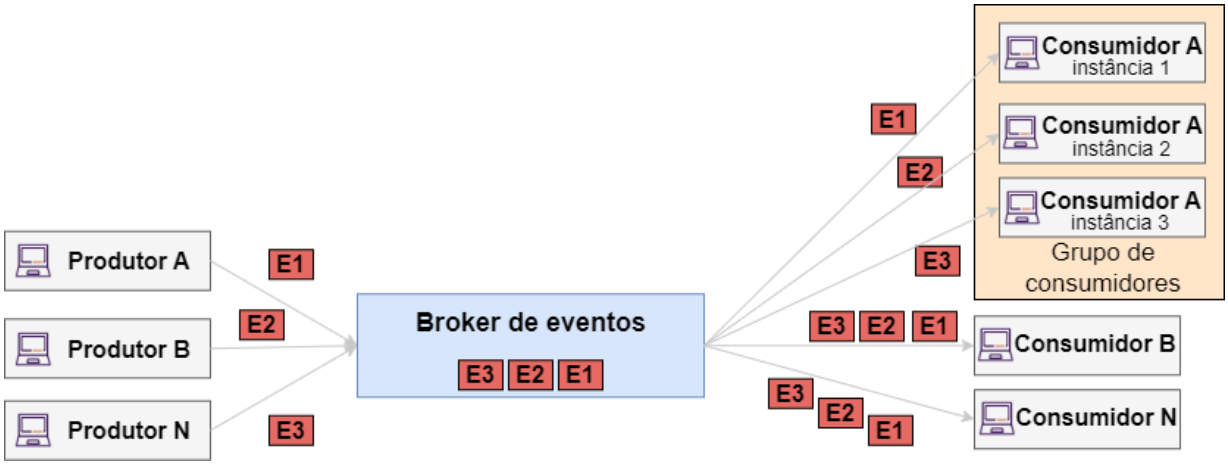

Figura 3.5: Entrega de evento pub/sub muitos-para-muitos.

Um exemplo de uma possível solução que usaria esse padrão é um sistema de e-commerce. Quando um evento de pagamento negado ocorrer, várias aplicações consumidoras poderiam receber uma cópia para fazer sua tarefa. Por exemplo, uma cópia para a aplicação que atualiza a situação do pedido e uma cópia para a aplicação que notifica o cliente de que seu pagamento foi negado.

Event streaming

A comunicação no padrão *event streaming* **refere-se a quando executamos um grande fluxo contínuo de eventos de série**. Eventos de série é um tipo de evento, conforme falamos na seção *Tipos de mensagem* do capítulo 2. *Entendendo os principais conceitos*. Esse padrão é uma variação do padrão *publish/subscribe*.

O *broker* de eventos que suporta esse padrão é capaz de armazenar uma sequência ordenada de eventos que são continuamente anexadas (*append-only*) no final do tópico ou log. Esses eventos podem ser reproduzidos na sequência pelos consumidores a partir de qualquer posição do log.

Figura 3.6: Entrega de evento de série.

Em uma plataforma de *streaming*, esses tipos de eventos passam por diversas operações para fazerem sentido, diferentemente dos eventos do tipo discreto. Essas operações podem ser de filtro (*filtering*), combinação (*combining*), divisão (*splitting*) e mapeamento (*mapping*).

Existem vários fornecedores de *broker* que suportam esse padrão, sendo que os mais conhecidos são o Apache Kafka, o Confluent, o Amazon Kinesis e o Microsoft Event Hubs.

Um exemplo de uma possível solução que usaria esse padrão é um sistema de detecção e prevenção de fraudes, no qual o sistema precisa executar diversas operações em cima de um grande fluxo contínuo de dados/eventos em movimento (*Data in motion*).

Request/reply

A comunicação no padrão *request-reply* **refere-se a quando o produtor da mensagem precisa de uma resposta em algum momento no futuro**. Nesse padrão, utilizamos dois canais *point-to-point* do tipo fila: um para solicitações e outro para respostas.

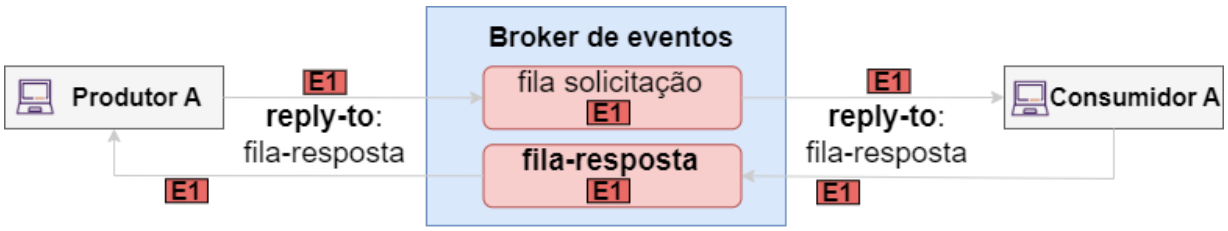

Figura 3.7: Entrega de evento request-reply.

A aplicação produtora publica o evento ou uma mensagem e informa, através do cabeçalho *reply-to*, de qual fila ela quer receber a resposta. O *broker* de eventos faz o roteamento para a aplicação consumidora, repassando esse cabeçalho. Após o processamento, as aplicações invertem seus papéis: a aplicação consumidora publica o evento de resposta para o canal especificado no cabeçalho *reply-to* e o *broker* de eventos encaminha para a aplicação produtora, agora fazendo o papel de consumidora.

Esse padrão pode ter a variação de *um-para-muitos*, em que cada consumidor publicará na mesma fila indicada no cabeçalho *reply-to*.

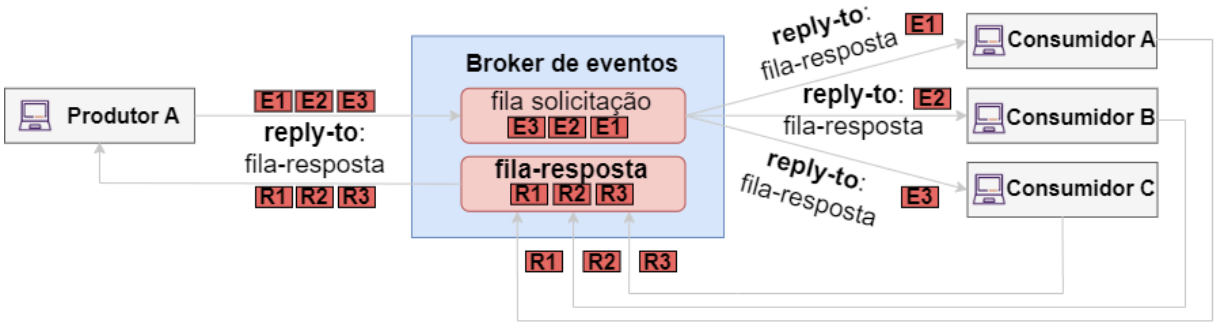

Figura 3.8: Entrega de evento request-reply um-para-muitos.

Esse padrão difere bastante dos outros três que vimos anteriormente. Dentro da nossa solução EDA, conseguimos simular uma comunicação síncrona *request-response* de forma assíncrona através do *broker* de eventos. É ideal para casos onde o processamento da resposta pode levar bastante tempo, não bloqueando a aplicação produtora.

Vale ressaltar que esse padrão não substitui o modelo tradicional *request-response*, onde um usuário ou uma aplicação precisa manter a conexão aberta aguardando uma resposta imediata dentro de um tempo adequado.

Um exemplo de um possível uso desse padrão seria em uma solução com fluxo ou coreografia de eventos. Uma aplicação produtora precisa buscar uma lista de entidades de negócio e, com a resposta,

seguir com seu processamento e/ou iniciar um novo fluxo de eventos para cada entidade da lista.

A comunicação assíncrona através de mensagens ou eventos possibilita que o produtor não precise aguardar até que o evento seja entregue para o consumidor. Em vez disso, ele pode seguir seu fluxo de processamento assim que o *broker* aceitar o evento. Esse comportamento segue o princípio do padrão *fire-and-forget*, onde o produtor literalmente publica o evento e o "*esquece*".

3.2 Padrões de comunicação de mudança de estado

Os padrões de comunicação de mudança de estado referem-se à forma como comunicamos e armazenamos uma mudança de estado de uma entidade de negócio. Para comunicar uma mudança de estado, precisamos decidir o quanto de informação vamos transmitir no corpo do evento.

Por exemplo, quando uma aplicação produtora publica um evento de *pedido atualizado*, ela está informando que um pedido teve uma mudança em seu estado, por exemplo, uma atualização dos itens e valor do pedido. Qual seria o conteúdo do evento do *pedido atualizado*? Apenas o identificador do pedido? Apenas o campo que realmente foi atualizado, juntamente com seu identificador? Toda a informação do pedido? Ou, ainda, toda informação antes da atualização e depois da atualização?

Nesta seção, vamos explorar alguns desses padrões e entender as suas vantagens e desvantagens.

Notificação de evento

O padrão de notificação de evento (*Event notification*) é o mais comum em uma solução de EDA. **A aplicação produtora notifica através de eventos, com um mínimo de informação, que alguma mudança de estado ocorreu**, e aplicações consumidoras interessadas ficam cientes da mudança.

O corpo dos eventos são leves e pequenos, pois carregam apenas informações básicas da entidade de negócio. Esse padrão é ideal quando precisamos reduzir o tamanho do corpo do evento para melhorar o desempenho do fluxo de publicação e consumo, pois eventos pequenos são mais fáceis de processar e mais rápidos em termos de serialização e desserialização.

Em uma solução hipotética de e-commerce, um evento de *pedido atualizado* teria apenas o identificador do pedido no seu corpo para comunicar uma mudança de estado.

```
{
    "orderId": "6e444fb5-f5c8-4d55-91eb-04b3abadf64b"
}
```

E se uma aplicação consumidora precisar de mais informação do pedido para realizar a sua tarefa além do identificador do pedido? O que ocorre é que a aplicação consumidora precisa realizar uma chamada extra para buscar as informações adicionais da entidade de negócio. Isso normalmente acontece através de uma chamada síncrona *request-response* na aplicação produtora, perdendo todo o benefício do mínimo acoplamento entre as aplicações através de eventos. Mesmo se usarmos o modelo *request-reply*, que vimos na seção anterior, para buscar as informações, ainda assim temos uma chamada extra.

A seguir, vamos ver uma solução hipotética de um sistema de e-commerce. Nela, temos uma aplicação produtora, `pedido`, que tem todas informações do pedido e alguns dados do cliente relacionado, e duas aplicações consumidoras: `envio`, responsável por entregar o

pedido no endereço do cliente, e `notificacao`, responsável por enviar um e-mail para o cliente quando seu pedido for atualizado.

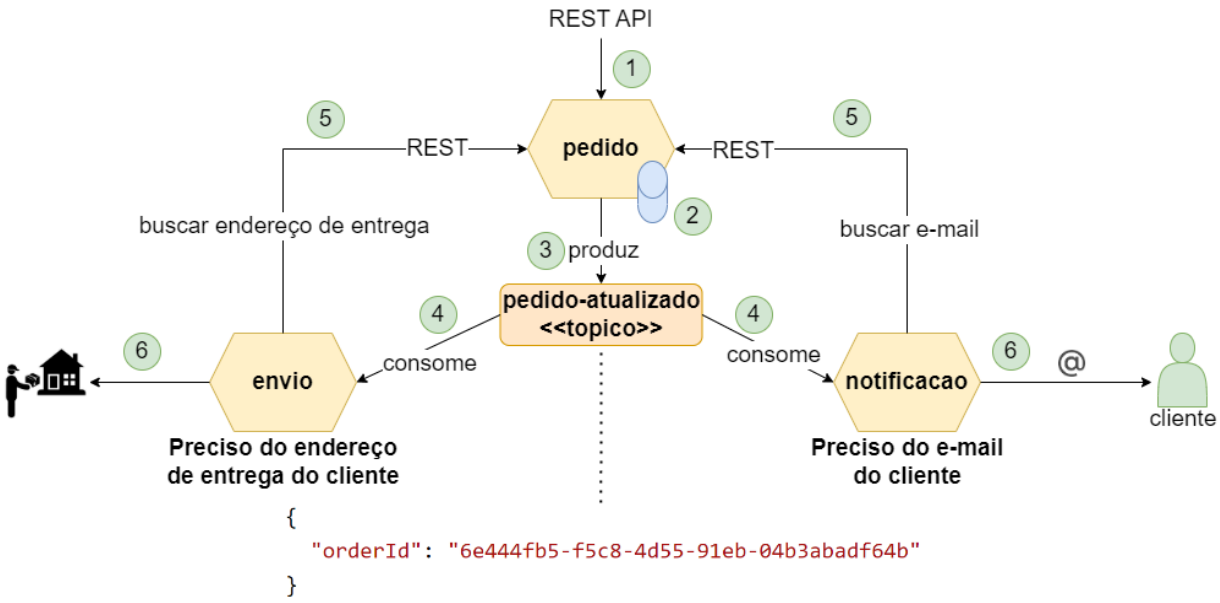

Figura 3.9: Exemplo de solução de notificação de evento.

1. Sistema atualiza um determinado pedido;
2. A aplicação `pedido` faz as devidas validações e atualiza o pedido na sua base de dados;
3. A aplicação `pedido` publica o evento `pedido-atualizado`;
4. As aplicações `envio` e `notificacao` consomem o evento `pedido-atualizado`;
5. As aplicações `envio` e `notificacao` buscam informações adicionais na aplicação `pedido` através de uma chamada REST;
6. A aplicação `envio` envia o pedido para o endereço do cliente e a aplicação `notificacao` envia o e-mail para o cliente.

A natureza desacoplada da EDA faz com que a aplicação produtora não tenha conhecimento nenhum da aplicação consumidora. Nesse padrão, existe uma grande possibilidade de as aplicações consumidoras serem acopladas com a aplicação produtora, por precisarem buscar informações adicionais da entidade de negócio.

Essa decisão pode aumentar a complexidade e impactar no desempenho da solução como um todo, por precisar de chamadas remotas extras, e na resiliência, por necessitar que a aplicação, dona da informação, esteja sempre on-line, além de sobrecarregá-la.

Event-carried state transfer

O padrão *Event-carried state transfer* é uma variação do padrão de notificação de evento em que **a aplicação produtora notifica, através de eventos, com todas as informações da entidade de negócio ou uma parte relevante dela, que alguma mudança de estado ocorreu** e as aplicações consumidoras interessadas ficam cientes da mudança.

O corpo dos eventos normalmente carrega todas as informações da entidade de negócio. Esse padrão resolve o problema do padrão de *notificação de evento*, por remover a necessidade de chamadas remotas extras das aplicações consumidoras para buscar informações adicionais da entidade de negócio; mas passamos a lidar com outros tipos de problemas, além do impacto no desempenho do fluxo de publicação e consumo.

Como todas as informações da entidade de negócio estão no corpo do evento, temos algumas implicações relacionadas à privacidade dos dados. Devemos garantir que nosso fluxo de publicação e consumo de eventos esteja de acordo com as políticas e regulamentações.

Não existe uma regra em relação à quantidade de informações que vamos transmitir no corpo do evento. Podemos modelar nosso esquema com as informações completas da entidade de negócio (imagina uma entidade de negócio que tenha mil campos, por exemplo) ou podemos colocar parte das informações que possam atender às aplicações consumidoras (pelo menos as que podemos controlar dentro do nosso escopo) sem expor dados desnecessariamente.

A definição do esquema faz parte do design e essa definição pode afetar diretamente a solução como um todo.

Continuando nosso exemplo hipotético de uma solução de e-commerce, vamos adicionar algumas outras informações básicas da entidade de negócio no corpo do evento *pedido atualizado*.

Como vamos modelar sem as informações completas da entidade de negócio, para aproveitar os benefícios dos dois padrões, a ideia é definirmos nosso esquema com o mínimo de informações possível, mas prevendo quais informações outras aplicações precisam, dentro da nossa solução, evitando chamadas remotas extras.

```
{
    "orderId": "6e444fb5-f5c8-4d55-91eb-04b3abadf64b",
    "totalAmount": 250.00,
    "status": "approved",
    "deliveryAddress": {
        "street": "Rua Presidente Vargas",
        "number": "1000",
        "postalCode": "24300-000"
    },
    "contact": {
        "email": "teste@gmail.com"
    }
}
```

Após a remodelagem do esquema do nosso evento de atualização de pedido, nossa solução ficou menos complexa e sem chamadas remotas extras para buscar informações adicionais de pedido.

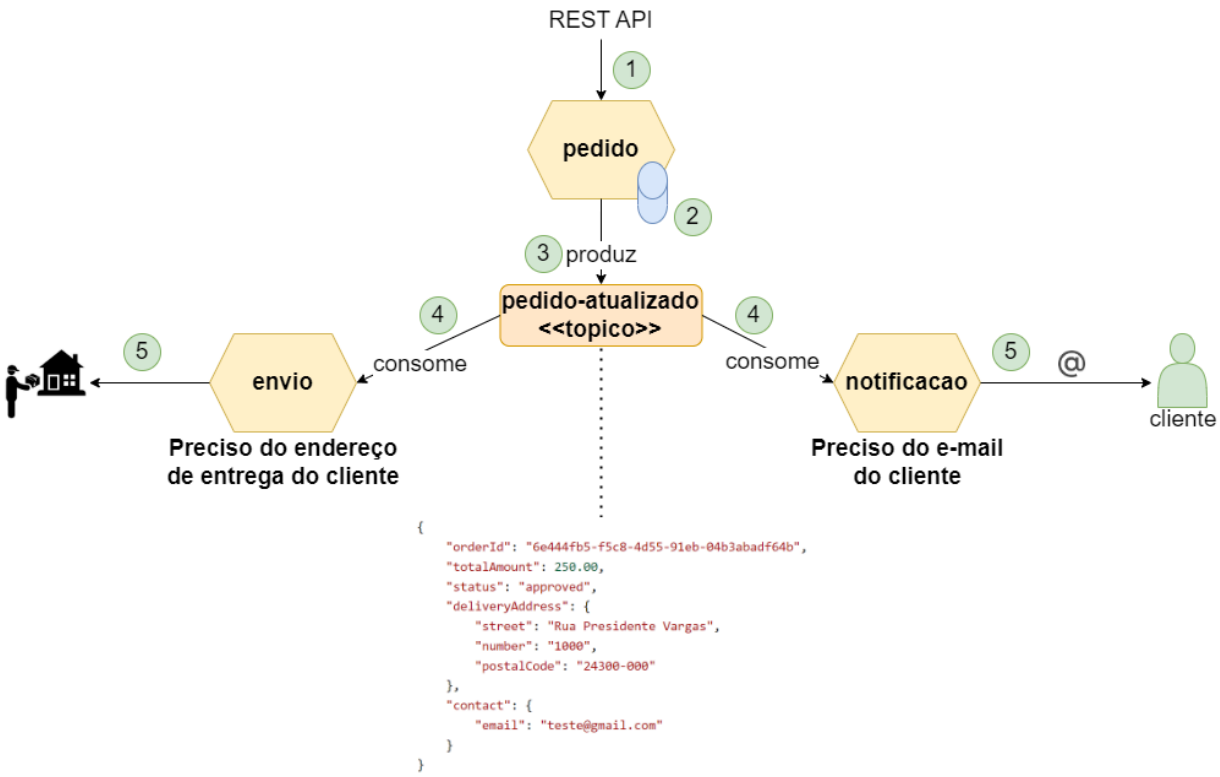

Figura 3.10: Exemplo de solução de notificação de evento sem chamadas remotas extras.

1. Sistema atualiza um determinado pedido;
2. A aplicação `pedido` faz as devidas validações e atualiza o pedido na sua base de dados;
3. A aplicação `pedido` publica o evento `pedido-atualizado` ;
4. As aplicações `envio` e `notificacao` consomem o evento `pedido-atualizado` ;
5. A aplicação `envio` envia o pedido para o endereço do cliente e a aplicação `notificacao` envia o e-mail para o cliente.

Nessa abordagem de usarmos parte da entidade de negócio para termos os benefícios dos dois padrões, pode ocorrer de o nosso esquema atual não atender novas aplicações consumidoras adicionadas na solução. Nesse caso, essas aplicações farão uma chamada remota extra para buscar informações adicionais, a menos que atualizemos nosso esquema.

Alterar o esquema implica em alteração de contrato. Mesmo se adicionarmos apenas campos novos opcionais para atender novas aplicações consumidoras, na teoria não é para impactar os consumidores existentes — seria uma evolução ou versionamento com *compatibilidade com versão anterior*, mas nem sempre é o que acontece na prática.

Claim checks

O padrão de *claim checks* é parecido com o padrão de *notificação de eventos*, porém com a adição de um novo componente na solução: um serviço externo, como um banco de dados. **A aplicação produtora armazena toda a informação da entidade de negócio em um serviço externo e notifica o *broker*, com um identificador (o *claim check*) no corpo do evento, de que alguma mudança de estado ocorreu**. As aplicações consumidoras interessadas ficam cientes da mudança e realizam uma chamada remota no novo serviço externo para buscar as informações da entidade de negócio utilizando o identificador.

Esse padrão é ideal nos casos de uso onde o corpo do evento for muito grande. Mesmo se utilizarmos um formato binário, como o Apache Avro ou ProtoBuf, que são formatos compactos, podemos enfrentar grandes quantidades de informações. Eventos muito grandes não são recomendados, pois afetam o desempenho e o custo da solução. Eles exigem um maior poder computacional do *broker* e geram uma alta latência de rede. O custo tende a ser mais caro do que armazenar as informações em um serviço externo, como um banco de dados.

Outro fator importante, e que vamos abordar nos próximos capítulos, é que cada fornecedor de *broker* suporta até um determinado limite de tamanho de evento. Com isso, impossibilita a transmissão de determinados conteúdos no corpo dos eventos, o que torna a implementação desse padrão obrigatória.

Dando sequência no nosso exemplo hipotético de uma solução de e-commerce, vamos remodelar nossa solução com o padrão *claim check*.

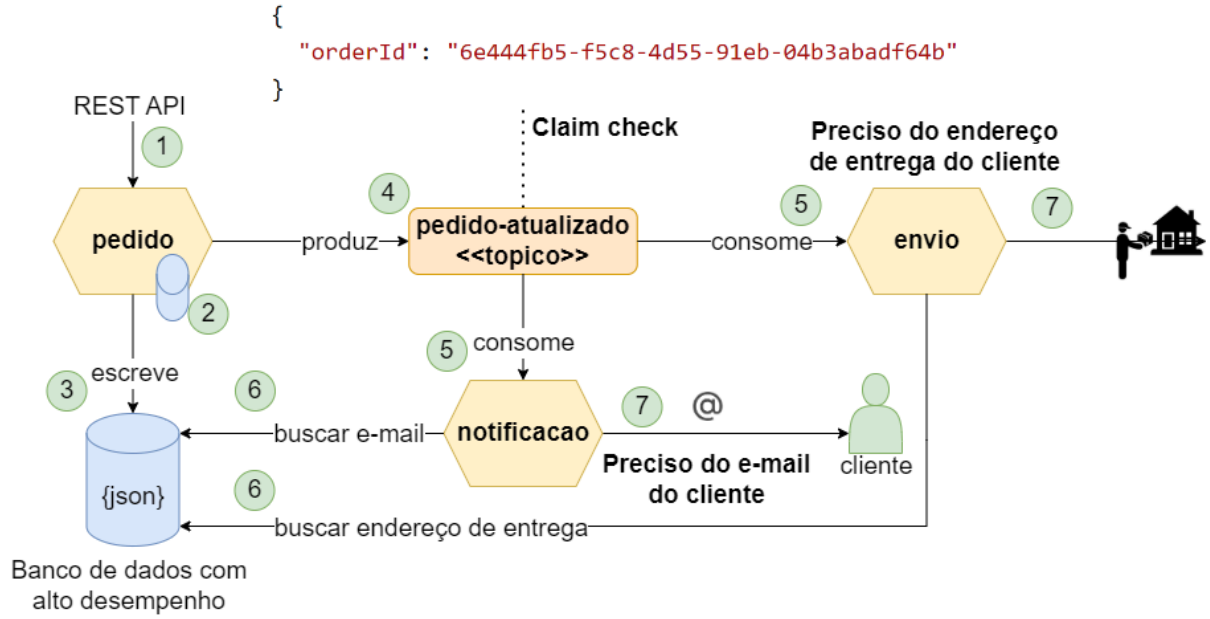

Figura 3.11: Exemplo de solução de com o padrão claim check.

1. Sistema atualiza um determinado pedido;
2. A aplicação `pedido` faz as devidas validações e atualiza o pedido na sua base de dados;
3. A aplicação `pedido` grava todo o conteúdo do pedido na base de dados externa;
4. A aplicação `pedido` publica o evento `pedido-atualizado` ;
5. As aplicações `envio` e `notificacao` consomem o evento `pedido-atualizado` ;
6. As aplicações `envio` e `notificacao` buscam informações adicionais no banco de dados externo;
7. A aplicação `envio` envia o pedido para o endereço do cliente e a aplicação `notificacao` envia o e-mail para o cliente.

Esse padrão não sobrecarrega a aplicação produtora, como no padrão de *notificação de evento*, porém continuamos com mais uma chamada remota na solução.

Apesar de o nosso exemplo anterior ser válido, raramente a informação desse tipo de entidade de negócio, como um pedido, vai atingir o limite de tamanho de evento de um *broker*. Nesse cenário, a recomendação é tentar aproveitar os benefícios dos padrões de *notificação de eventos* e *event-carried state transfer* vistos anteriormente.

Um caso de uso muito comum em que o padrão *claim check* funciona bem é quando precisamos transmitir e manipular arquivos do tipo imagens, áudio, vídeo ou documentos de texto diversos. Esses tipos de arquivos podem facilmente ultrapassar o limite dos *brokers* de eventos.

A seguir, vamos ver um exemplo simplificado de uma possível solução EDA de processamento de imagens utilizando o padrão *claim check* e *publish-subscribe*. Essa solução contempla uma aplicação produtora e três consumidoras, além do *broker* de eventos por meio do tópico *imagem-carregada* e um banco de dados do tipo *blob* (https://pt.wikipedia.org/wiki/BLOB), como o Amazon S3, Azure Blob Storage e Google Cloud Storage.

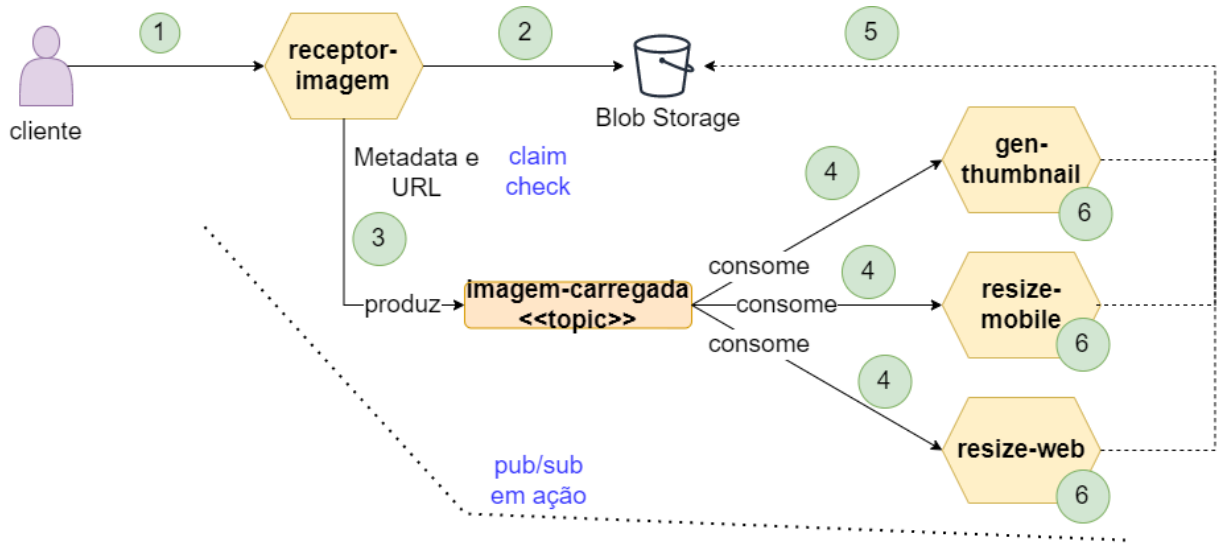

Figura 3.12: Exemplo de solução de processamento de imagem com o padrão claim check e pub/sub.

1. Cliente faz o upload da imagem no sistema;

2. A aplicação `receptor-imagem` faz as devidas validações e faz o upload da imagem no banco de dados;
3. A aplicação `receptor-imagem` publica o evento `imagem-carregada` com os metadados e a URL da imagem;
4. As aplicações `gen-thumbnail`, `resize-mobile` e `resize-web` consomem o evento `imagem-carregada`;
5. As aplicações `gen-thumbnail`, `resize-mobile` e `resize-web` busca a imagem pela sua URL;
6. As aplicações são responsáveis por:
    - `gen-thumbnail` – gerar o *thumbnail* da imagem;
    - `resize-mobile` – redimensionar a imagem para dispositivos mobile;
    - `resize-web` – redimensionar a imagem para web.

O resultado das três aplicações consumidoras será um novo upload das imagens após a manipulação (omitido no desenho para simplificar) para possíveis aplicações interessadas – por exemplo, uma aplicação responsável por colocar marca d'água nas imagens redimensionadas para web.

Event Sourcing

Tradicionalmente, utilizamos banco de dados em nossas aplicações para armazenar o estado atual de uma determinada entidade de negócio. Por exemplo, se o pedido estiver com o status *entregue*, armazenamos na base de dados apenas as informações do pedido nesse estado; ou, quando atualizamos alguma informação do pedido, apenas os dados após a atualização serão mantidos na base de dados. Esse comportamento atende a muitos casos de uso.

Tabela Pedido				
pedido ID	produto ID	quantidade	status	(...)
12345	900	10	criado	...

Após atualização →

Tabela Pedido				
pedido ID	produto ID	quantidade	status	(...)
12345	900	**6**	**confirmado**	...

Figura 3.13: Exemplo de tabela de pedido após uma atualização.

Mas às vezes, além do estado atual, precisamos de todo o histórico das ações que ocorreram na entidade. Imagine um sistema onde o cliente precisa acompanhar o rastreamento do seu pedido. O sistema precisa exibir para o cliente não somente onde o pacote referente ao seu pedido está naquele momento, mas também todo o histórico por onde o pacote passou na linha do tempo.

Figura 3.14: Exemplo de rastreamento de um pedido na amazon.com.

Event sourcing **é um padrão arquitetural onde mantemos toda a sequência de ações de mudança do estado de uma entidade de negócio através de eventos**. Isso permite que as aplicações reconstruam a entidade de negócio desde a sua criação.

Imagine, no nosso exemplo hipotético de uma solução de e-commerce, que um determinado pedido possa ter a seguinte sequência de eventos na linha do tempo, até que ela seja entregue/fechada:

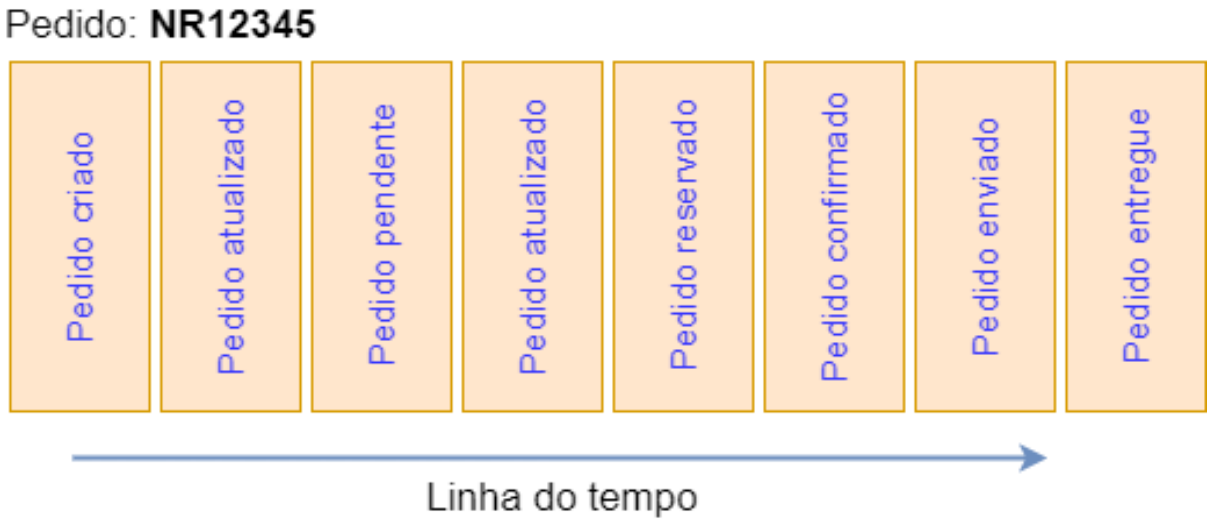

Figura 3.15: Exemplo de uma sequência de eventos de uma solução de e-commerce simplificada

Cada evento é uma mudança de estado, portanto, cada novo evento só pode ser anexado ou armazenado na sequência, termo conhecido como *append-only*. Esse padrão se encaixa naturalmente com EDA, por se tratar de eventos.

Tabela Pedido					
numero pedido	data	evento	versão	corpo/ payload	(...)
NR12345	10/11/2022 11:00	PedidoCriado	1.0	{}	...
NR12345	10/11/2022 11:10	PedidoAtualizado	1.0	{}	...
NR12345	10/11/2022 11:12	PedidoPendente	1.0	{}	...
NR12345	10/11/2022 11:20	PedidoAtualizado	1.0	{}	...
NR12345	10/11/2022 11:21	PedidoReservado	1.0	{}	...
NR12345	10/11/2022 11:25	PedidoConfirmado	1.0	{}	...
NR12345	10/11/2022 12:00	PedidoEnviado	1.0	{}	...
NR12345	12/11/2022 10:20	PedidoEntregue	1.0	{}	...

Figura 3.16: Exemplo tabela de pedido usando o padrão Event Sourcing.

Nesse nosso exemplo, a solução conseguiria responder a estas perguntas:

1. Qual é a situação atual do pedido NR12345? e
2. Qual o histórico do pedido NR12345 desde a sua criação? Seja para fins de auditoria, ou alguma razão legal pela qual não se

pode excluir registro, ou para disponibilizar uma funcionalidade de rastreio.

O uso deste padrão é **recomendado quando a solução tem requisitos de rastrear toda sequência de eventos de uma entidade de negócio, reverter alterações a partir de um ponto específico ou manter histórico e/ou registro de auditoria**.

Um exemplo de sistema que poderia usar esse padrão é um sistema de controle de versão como o Git, onde cada operação de *commit* representa um evento e todo o histórico é mantido, com a possibilidade de reverter a qualquer momento.

Event sourcing só é possível quando utilizamos o padrão *event-carried state transfer*, pois precisamos armazenar a sequência dos eventos com as informações completas da entidade de negócio.

Podemos implementar esse padrão utilizando o próprio *broker* de eventos em vez de um banco de dados externo. Temos que garantir que o *broker* de eventos tenha capacidades suficientes de armazenamento, para mantermos o estado atual e o histórico da entidade de negócio, além de ser *append-only* (apenas insere no final da sequência) e permitir que as aplicações consumidoras consigam processar a sequência dos eventos a partir de qualquer ponto. Nos capítulos seguintes sobre *broker* de eventos, vamos entender melhor as capacidades dos *brokers* de mercado e quais podem atender aos requisitos deste padrão.

Em um cenário onde temos centenas de eventos de uma determinada entidade de negócio, reprocessar todo o histórico pode impactar o desempenho da solução. Uma opção seria criar periodicamente um *snapshot* com o estado mais atual da entidade. Para reconstruí-lo, a aplicação reprocessa os eventos a partir do *snapshot* mais recente, resultando em poucos eventos para processar.

Para sistemas pequenos que envolvem domínio de negócio simples, ou se a solução não contemplar os requisitos que mencionamos anteriormente, a implementação desse padrão não é recomendada.

Event sourcing adiciona dois grandes desafios na solução:

1. A evolução ou versionamento do esquema dos eventos;
2. A complexidade e ineficiência na consulta (*query*) dos dados.

Os esquemas dos eventos são evoluídos e versionados com o tempo. Imagine que parte da sequência de eventos utiliza um esquema obsoleto. Nesse caso, não seria possível processar toda a sequência de eventos a menos que a aplicação consumidora seja projetada para interpretar o esquema antigo também.

Realizar consultas e filtros no corpo de cada evento dentro de uma sequência é complexo e ineficiente. Mesmo havendo a alternativa de criação de *snapshots* para mitigar o problema da ineficiência, ainda assim aumentamos a complexidade da solução, pois teremos esforço extra para criar e gerenciar os *snapshots*.

Para mitigar a complexidade e ineficiência da consulta e filtro dos dados e de alguma forma a evolução ou versionamento de esquema, usamos o padrão CQRS (*Command Query Responsibility Segregation*) em conjunto com o *event sourcing*.

3.3 CQRS (Command Query Responsibility Segregation)

Em uma arquitetura tradicional, quando desenvolvemos nossa aplicação (ou microsserviço), normalmente ela suporta todas as operações de *criação* (*create*), *leitura* (*read*), *atualização* (*update*) e *remoção* (*delete*) — o famoso CRUD. O mesmo modelo de dados é usado em ambas operações, de escrita (criação, atualização e

remoção) e leitura. Esse tipo de arquitetura resolve muitas das necessidades do negócio.

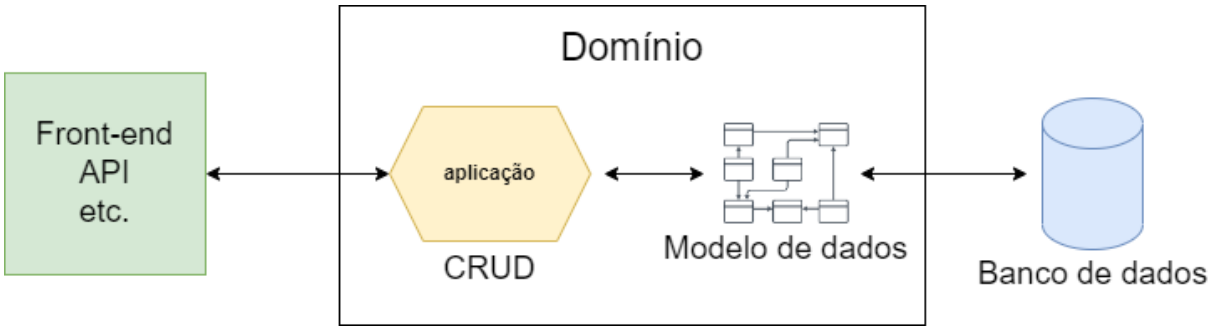

Figura 3.17: Arquitetura tradicional compartilhando o mesmo modelo de dados.

Mas e quando nosso negócio exige uma carga de trabalho muito maior na leitura dos dados? Onde a leitura dos dados têm requisitos de desempenho e escalabilidade distintos em comparação à escrita dos dados? E se o nosso modelo de dados (*data model*) for complexo e pouco performático para consultar informações para o negócio?

O padrão CQRS (*Command Query Responsibility Segregation*) **refere-se a quando separamos as nossas operações de escrita e leitura em diferentes modelos de dados, aplicações e opcionalmente banco de dados**. Usamos *comandos* para escrita dos dados e *queries* para leitura dos dados.

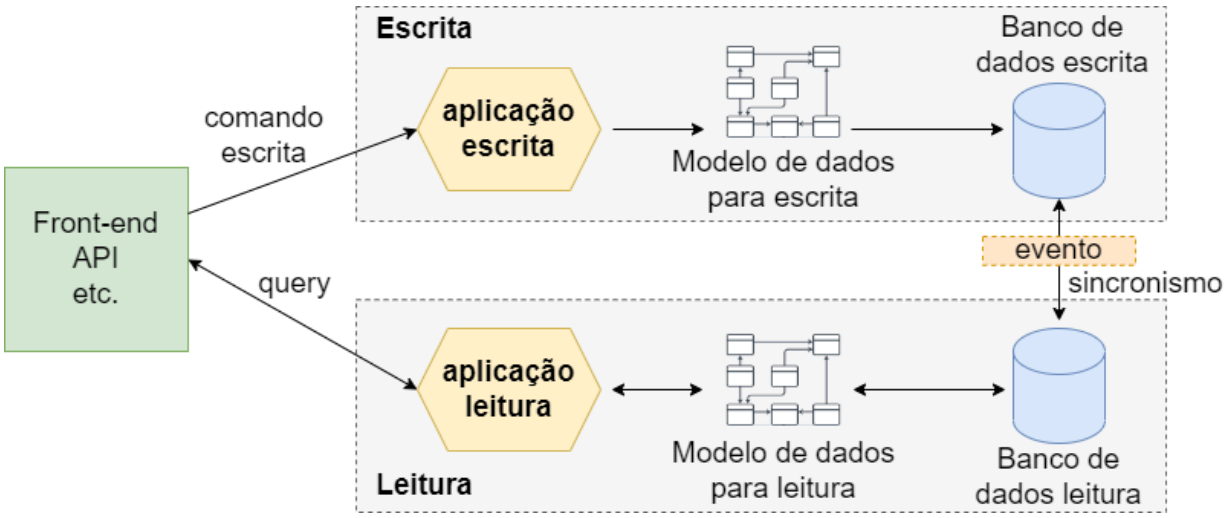

Figura 3.18: Exemplo da arquitetura com CQRS – aplicação, modelo e banco de dados distintos para cada operação.

Com modelos distintos, podemos otimizá-los de acordo com o seu propósito, e com banco de dados e aplicações distintas não é diferente: podemos projetá-los com a tecnologia mais apropriada e atender a diferentes requisitos não funcionais para cada propósito.

A consistência eventual é outro fator importante quando separamos os banco de dados. O sincronismo entre as bases de dados ocorre através de eventos. Clientes podem ler dados inconsistentes até que o sincronismo seja finalizado na parte da leitura.

Implementar esse padrão adiciona uma maior complexidade na solução, pois estamos inserindo mais componentes e interações. Mas essa complexidade pode ser compensada pelos seus benefícios. Otimizar o modelo de dados de acordo com a necessidade de escrita ou leitura pode proporcionar um melhor desempenho na leitura e minimizar a duplicação de dados na escrita através da normalização do modelo.

Escolher a melhor tecnologia de acordo com o propósito também é um benefício. Por exemplo, podemos usar uma base de dados SQL na parte da escrita para garantir a consistência e usar uma base de dados na parte de leitura com um melhor desempenho para consultas complexas, como o ElasticSearch. Além disso, podemos

escalar a aplicação de acordo com a necessidade de leitura ou escrita. A tendência é termos muito mais leitura do que escrita na solução.

Como no *event sourcing*, para sistemas pequenos que envolvem domínio de negócio simples não é recomendado utilizar o CQRS, pois vai gerar mais confusão do que solução. Nesse caso, a estratégia do CRUD atenderia muito bem.

A seguir, vamos mostrar um exemplo simples de solução de como poderíamos trabalhar com os dois padrões, *event sourcing* e CQRS, para mitigar os problemas/desafios do *event sourcing* que comentamos anteriormente.

Event Sourcing e CQRS

O desenho a seguir é o resumo de uma possível solução utilizando ambos os padrões. Como você pode imaginar, no lado da escrita, utilizamos o padrão *event sourcing* e, no lado da leitura, podemos desnormalizar nosso modelo e disponibilizar uma *visualização materializada* (em inglês, *materialized view*. Saiba mais em: https://en.wikipedia.org/wiki/Materialized_view). Essa abordagem pode nos ajudar a mitigar os desafios do padrão *event sourcing* mencionados anteriormente.

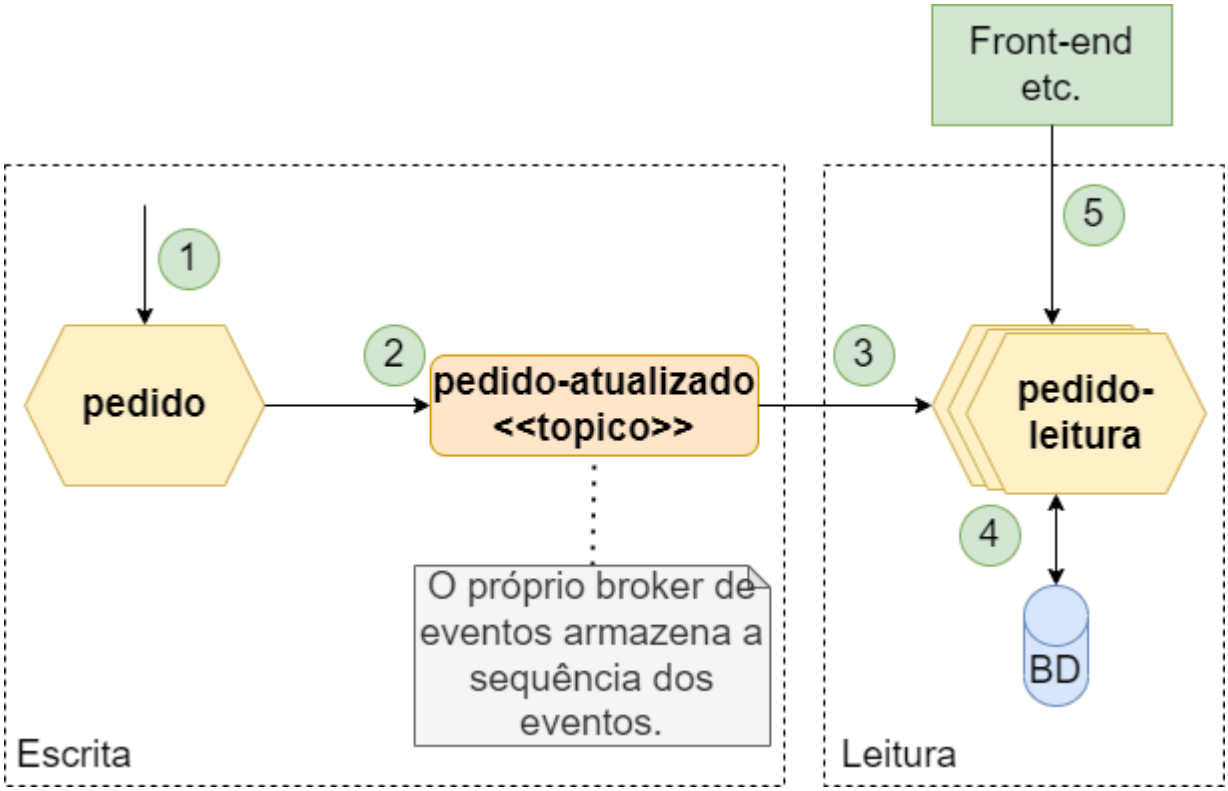

Figura 3.19: Exemplo de uma possível solução simplificada usando event sourcing e CQRS

1. Sistema atualiza um determinado pedido;
2. A aplicação `pedido` faz as devidas validações e publica o evento `pedido-atualizado`;
3. A aplicação `pedido-leitura` consome o evento `pedido-atualizado`;
4. A aplicação `pedido-leitura` cria uma visualização materializada de acordo com o modelo de leitura para otimizar as consultas;
5. O cliente faz as consultas apenas através da aplicação `pedido-leitura`.

Na verdade, não importa se na parte da escrita estamos usando o padrão de *event sourcing*, podemos usar o padrão CQRS do mesmo jeito para aproveitar seus benefícios.

Além disso, podemos utilizar uma outra abordagem onde precisamos unificar informações de três aplicações distintas, dentro da nossa coreografia de eventos, em um único modelo de dados para leitura e, assim, simplificar a consulta das informações e evitar consultas

complexas que podem afetar o desempenho da solução como um todo.

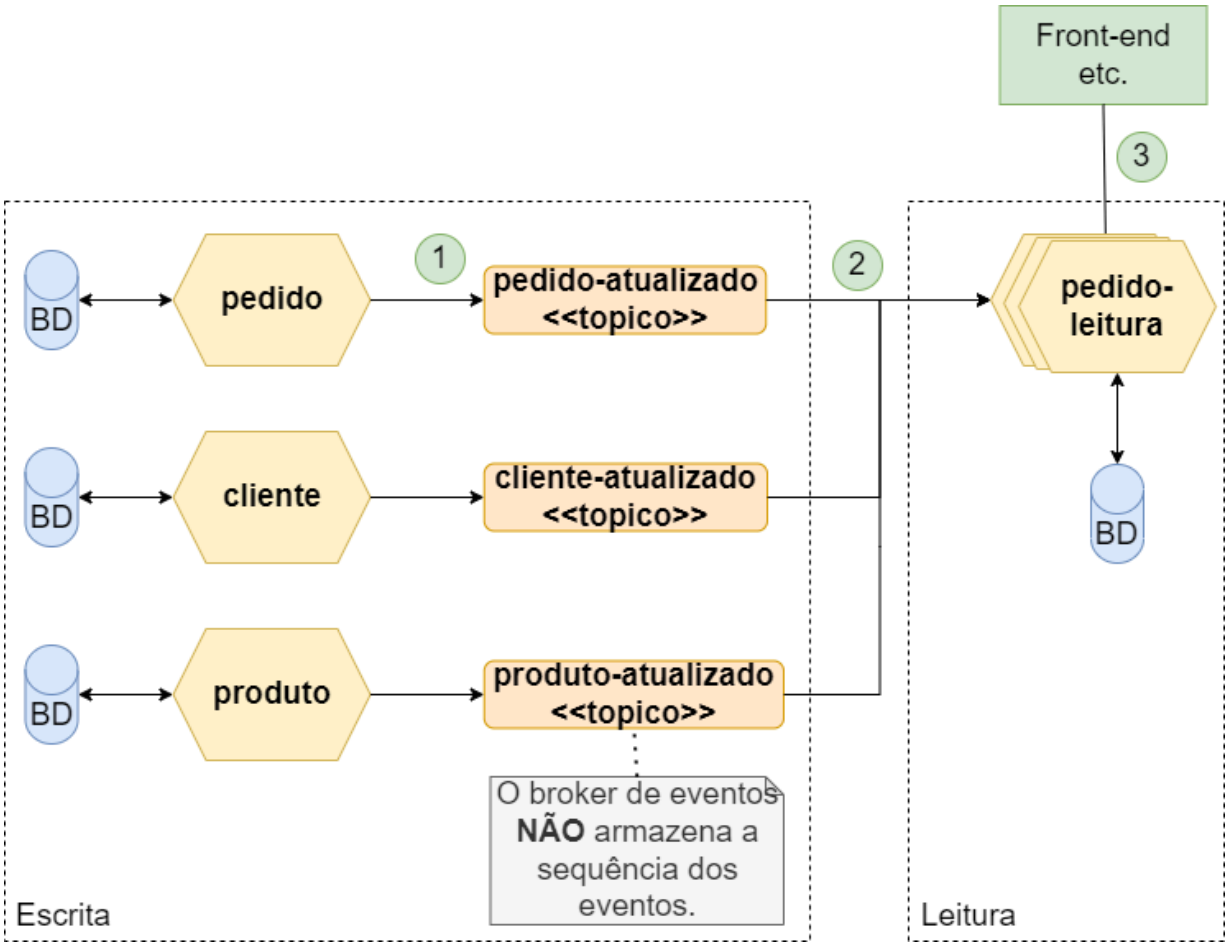

Figura 3.20: Exemplo de uma unificação dos dados de possível solução simplificada usando event sourcing e CQRS.

1. Sistema atualiza pedido, cliente ou produto;
2. A aplicação `pedido-leitura` consome diferentes eventos e constrói um modelo de dados desnormalizado;
3. O cliente faz as consultas apenas através da aplicação `pedido-leitura`.

Uma observação que vale destacar é que não devemos implementar os padrões *event sourcing* e/ou CQRS em todas aplicações ou microsserviços da nossa solução, mas apenas em um subconjunto de aplicações que atenda aos requisitos que mencionamos

anteriormente. Caso contrário, podemos gerar complexidades sem necessidade.

Além disso, todos os padrões que vimos nesta seção, como a *notificação de eventos*, *event-carried state transfer* e *claim check* podem e devem ser utilizados em conjunto na mesma solução de EDA de acordo com a necessidade.

3.4 Dead-letter queues (DLQ)

Em uma solução cada vez mais distribuída e desacoplada, é comum projetá-la para lidar com erros transitórios (problemas de rede e falhas de infraestrutura em geral), que podem causar indisponibilidades diversas durante um determinado período.

Podemos configurar políticas de retentativas, seja na perspectiva da aplicação produtora, para reenviar o evento para o *broker*, ou na perspectiva da aplicação consumidora, que pode consumir o mesmo evento novamente. Esses tipos de erros podem se resolver sozinhos (*self-healing*), sem nenhum tipo de intervenção humana.

Uma vez que os eventos chegaram ao *broker* e este confirmou o recebimento, ele passa a ter uma grande responsabilidade: a garantia de entrega dos eventos para as aplicações consumidoras registradas.

É um comportamento padrão dos *brokers* os eventos ficarem disponíveis no tópico de origem para consumo até que eles sejam confirmados pelas aplicações consumidoras como recebidos e/ou processados, podendo ser até removidos.

Mas e se, mesmo após todas as retentativas, as aplicações consumidoras não receberem os eventos ou não conseguirem confirmar seu recebimento após alguma falha de processamento? E se o evento expirar? — sim, podemos configurar um tempo máximo

de vida (*time-to-live*) do evento em uma fila ou tópico, como veremos nos capítulos seguintes, quando falarmos das capacidades do *broker*. E se o erro não for transitório (um problema nos dados em si por causa de um formato inválido), no qual realizar retentativas não adianta?

Podemos configurar filas especiais chamadas *Dead-letter queues* (DLQ). Todos os eventos que, por algum motivo, não conseguiram ser recebidos ou processados pelas aplicações consumidoras serão movidos para uma DLQ. Os fornecedores de *brokers* implementam esse padrão de forma nativa, permitindo que façamos configurações simples para habilitá-lo.

Imagine que, no nosso exemplo hipotético de e-commerce, as aplicações `envio` e/ou `notificacao` por algum motivo não conseguem consumir ou processar o evento `pedido-atualizado`. O que fazer com esses eventos que não foram entregues ou processados?

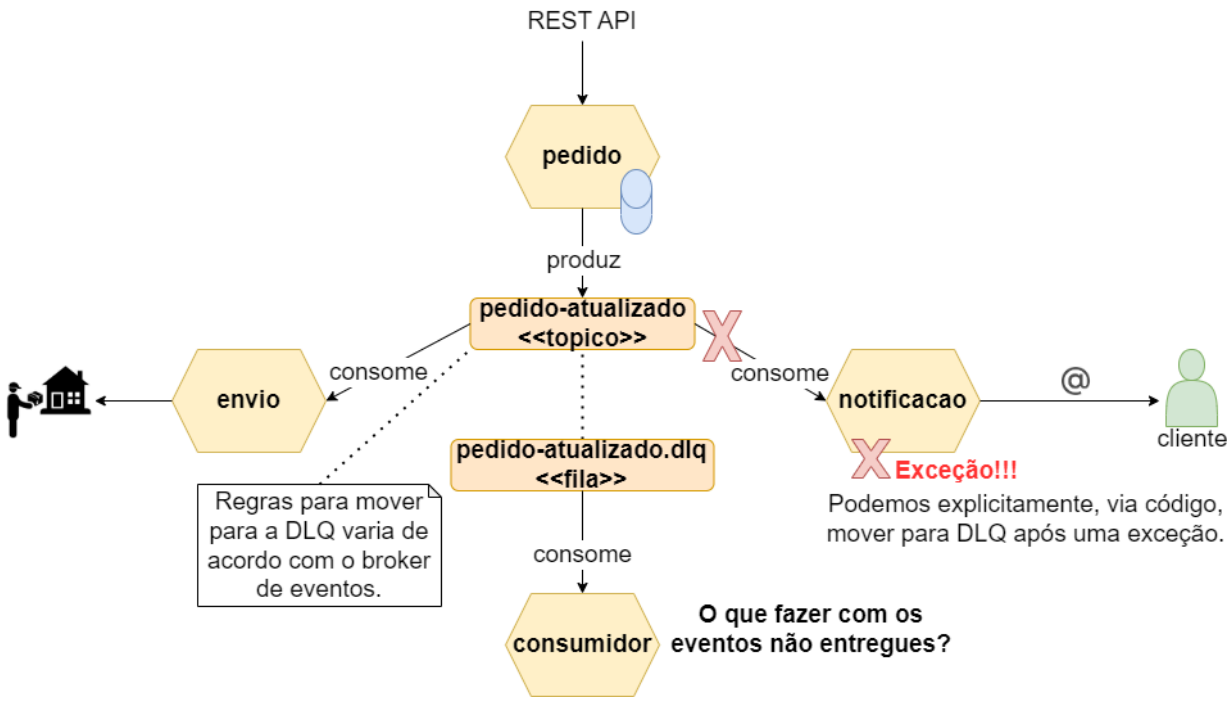

Figura 3.21: Exemplo genérico de uma solução com DLQ configurada.

O que fazer com esses eventos fica a cargo da necessidade do negócio. Podemos ter aplicações que consomem da DLQ e geram relatórios para alguma intervenção manual. Outras aplicações podem fazer algum tratamento automático e mover o evento para a fila de origem novamente para tentar ser reprocessado. As possibilidades são diversas.

No exemplo anterior, mostramos uma solução genérica de EDA com DLQ habilitada. Cada fornecedor de *broker* tem suas regras para mover um evento para uma DLQ, como eventos que ultrapassam o limite de tamanho, expiração do evento etc. Além disso, muitos deles permitem que a aplicação consumidora explicitamente, via código, mova o evento para a DLQ.

> A Uber compartilhou como eles utilizaram DLQ em sua solução de EDA. Veja em: https://www.uber.com/en-BR/blog/reliable-reprocessing.

Como a empresa ABC reprocessava os eventos movidos para a DLQ

Apesar de este ser um caso real, a empresa ABC é fictícia.

Alguns anos atrás, tínhamos um grande desafio: desenvolver uma solução com microsserviços e EDA para a empresa ABC, com grande potencial de crescimento de clientes e receita. A escolha dessas arquiteturas foi preparar o negócio da empresa para o crescimento. Todo seu *modus operandi* era através de seus sistemas legados e boa parte do processo era manual e custoso. Esse projeto correspondia a todo o processo de credenciamento do cliente, passando pela oferta, contrato, até a entrega do produto na casa do cliente.

De forma bem resumida, o cliente entrava no site (ou qualquer outro canal), se registrava (passava pelo processo de credenciamento) e depois realizava a compra pelo site. Esse processo de

credenciamento não era simples, tinha algumas etapas de enriquecimento dos dados e qualificação do cliente e, posteriormente, do pedido. Após a entrega do produto físico, era necessária a sua ativação. Toda a solução era feita através da coreografia de eventos com dezenas de microsserviços e eventos. A comunicação com os sistemas legados era síncrona utilizando `SOAP` e `XML`.

Não era viável migrar as aplicações monolíticas/legadas, devido à necessidade de um alto investimento de tempo e dinheiro. A abordagem foi fazer a nova solução se comunicar com os sistemas legados e com sistemas de terceiros através do padrão *adapters* ou *anti-corruption layer* (ACL) (https://learn.microsoft.com/pt-br/azure/architecture/patterns/anti-corruption-layer).

Como envolvia sistemas legados com pouco conhecimento dos dados e muitos cenários possíveis, ocorriam muitos erros semânticos no sistema legado. A solução não conseguia finalizar o processo e o operador ajustava os dados diretamente no legado. Isso ocasionava diversos problemas, como processo manual, a transação ficava com estado inconsistente na solução por ter que *finalizar* o fluxo diretamente no legado e por aí vai.

A solução

Os ACLs eram aplicações consumidoras que tinham a responsabilidade de comunicar com o sistema legado. Caso o sistema legado retornasse algum tipo de erro (normalmente através de códigos de erros diversos), os ACLs explicitamente, via código, moviam o evento para a DLQ correspondente.

Um microsserviço de manutenção sincronizava os eventos das DLQs com a base de dados de erros e depois os removia. O front-end era alimentado por essa base de dados, que tinha todas informações dos eventos, da DLQ, da fila ou tópico de origem e o motivo do erro.

O operador, através do front-end, que era uma tela de manutenção, interpretava o erro e alterava os dados (o corpo do evento)

manualmente e clicava no botão *reprocessar*. Essa ação fazia com que o evento voltasse para sua fila ou tópico de origem dentro da solução (coreografia de eventos), e o processo continuava.

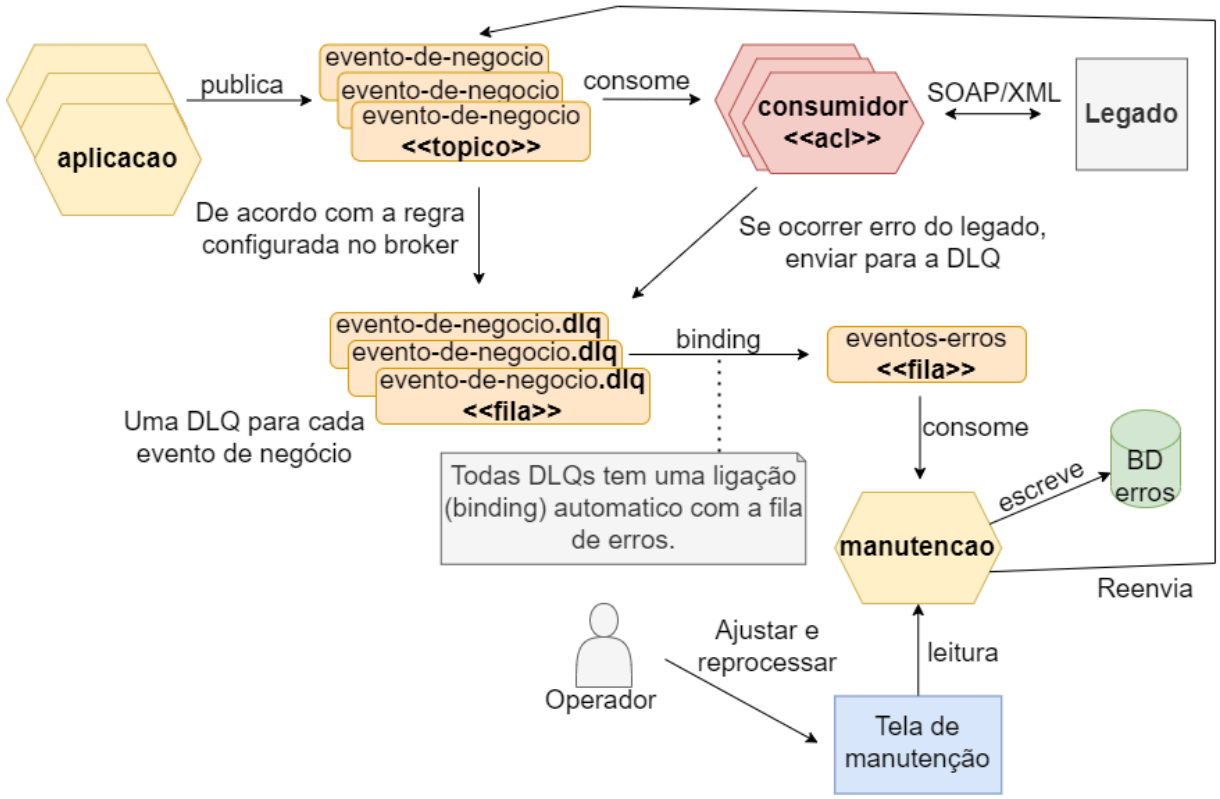

Figura 3.22: Como a empresa ABC reprocessava eventos da DLQ.

Todas as DLQs têm uma ligação (*binding*) automática com a fila de erros.

Essa abordagem foi boa para entendermos os erros que ocorriam na solução devido aos dados *inconsistentes* enviados ao legado e para permitir que o processo seguisse *por dentro* da solução. Mas era ruim no longo prazo, pois é uma solução que não escala e é propensa a erros, uma vez que dependia de uma intervenção humana para seguir com o processo. Ela ainda causava atrasos na entrega do pedido do cliente, além de aumentar a possibilidade de expor dados sensíveis pela tela de manutenção.

O diferencial foi que, a cada erro, além de uma intervenção rápida (mesmo que manualmente) e reenviar para reprocessar sem "marretada" no legado, era gerada uma demanda de ajuste na solução para adaptar a necessidade do legado. E assim, em pouco tempo, toda a solução se estabilizou, diminuindo consideravelmente os erros semânticos ocasionados por dados inconsistentes enviados ao legado.

Na época do projeto na empresa ABC, todo esse mecanismo foi desenvolvido. Atualmente, alguns fornecedores de *broker* oferecem, de algum modo, esse tipo de mecanismo, como o Service Bus da Azure (https://learn.microsoft.com/pt-br/azure/service-bus-messaging/service-bus-dead-letter-queues#sending-dead-lettered-messages-to-be-reprocessed) e o Amazon SQS (https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html#sqs-dead-letter-queues-redrive).

3.5 First in, first out (FIFO)

Em muitos dos cenários do fluxo de publicação e consumo de eventos que vimos até agora, nós não nos preocupamos com a ordem dos eventos que chegam à aplicação consumidora, mas há cenários onde a ordem dos eventos é crítica para o negócio.

Imagine que adicionamos uma aplicação ou microsserviço responsável pelo gerenciamento dos preços dos produtos no nosso exemplo hipotético de e-commerce. Essa aplicação altera o preço do `produto_A` para R$ 10. Alguns segundos depois, uma outra atualização de preço para o `produto_A`, agora passando para R$ 15. Qual desses dois eventos de atualização de preço a aplicação consumidora deveria receber por último? Fica clara a necessidade da garantia da ordem do evento nesse cenário.

Figura 3.23: Sequência de eventos respeitando a ordem de envio

A funcionalidade de *Primeiro a entrar, primeiro a sair*, ou, em inglês, *First in, first out*, ou simplesmente FIFO, **é a capacidade do *broker* de garantir que os eventos sejam consumidos na mesma ordem em que chegaram**. Basicamente, ela garante que:

1. A fila ou tópico deve receber os eventos na mesma ordem em que foram produzidos pela aplicação produtora;
2. A aplicação consumidora deve receber os eventos na mesma ordem em que eles foram recebidos na fila ou tópico.

O destino da mensagem do tipo fila naturalmente tem o princípio FIFO. Devido à natureza distribuída dos *brokers*, os fornecedores dos *brokers* utilizam o termo "*best-effort ordering*". Isso significa que eles fazem todo o esforço possível para garantir a ordem, mas, mesmo assim, ela pode não ser uma certeza.

Alguns *brokers* têm a funcionalidade que garante o princípio FIFO, mas precisamos levar em consideração o impacto no desempenho do fluxo de publicação e consumo dos eventos e no custo da

solução. O limite do número de eventos por segundo suportado pelo *broker* (*throughput*) pode diminuir e a *latência* (tempo entre a chegada do evento no *broker* até o recebimento do consumidor), aumentar. Normalmente esse tipo de funcionalidade em fornecedores de *brokers* no modelo Software as a Service (SaaS) tem um custo maior do que uma solução *best-effort ordering*. Por esses motivos, a utilização de uma fila do tipo FIFO tem que ser avaliada.

Voltando para o nosso cenário de atualização de preço, não importa a ordem do evento de atualização de preço para produtos diferentes. O que queremos garantir é a sequência dos eventos dentro de um contexto, no caso, um produto específico.

A sequência a seguir demonstra todos os eventos de atualização de preço de todos os produtos.

Figura 3.24: Sequência de eventos na fila de atualização de preço de produto.

Cada *broker* de eventos tem seu *modus operandi* para garantir a ordem dos eventos. De modo geral, a estratégia é criar grupos de acordo com o contexto.

Por exemplo, para garantirmos a ordem dos eventos relacionados ao `produto_A`, podemos relacionar todos os eventos do `produto_A` ao

mesmo grupo. No nosso exemplo, poderia ser o `produtoId`. Isso faz com que o *broker* garanta a ordem dos eventos do mesmo grupo.

Figura 3.25: Sequência de eventos na fila de atualização de preço de produto de acordo com o grupo.

Na imagem anterior, o *broker* garante que o evento do `produto_A` com valor R$ 50 será realmente entregue por último. Não importa a sequência dos outros produtos na perspectiva do `produto_A`.

A nomenclatura dessa divisão em grupos muda de acordo com o fornecedor, mas a ideia é a mesma. No Amazon SQS FIFO, é utilizada a propriedade **group_id** (https://docs.aws.amazon.com/pt_br/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues.html). Já no *Azure Service Bus*, é utilizada a propriedade **session_id** (https://learn.microsoft.com/pt-br/azure/service-bus-messaging/message-sessions). No Kafka, a ordem dos eventos é garantida dentro da partição (um tópico é feito por pelo menos uma partição). Para garantirmos que o *broker* redireciona o evento do mesmo contexto para a mesma partição, informamos uma chave na propriedade **key** ou **message_key** (https://developer.confluent.io/learn-kafka/apache-kafka/partitions/)

no momento da publicação do evento. O `produto_A` com ID `12345` cairá sempre na mesma partição.

3.6 SAGA

Um dos principais desafios de uma solução distribuída é manter a consistência do dado através de uma transação que abrange múltiplos serviços para atingir um objetivo de negócio. A natureza distribuída e com consistência eventual da EDA significa que é aceitável o dado ficar inconsistente durante um período; entretanto, o dado deve ficar consistente em algum momento. Isso é de extrema importância para qualquer negócio.

Imagine um processo de negócio de uma empresa de serviço de viagem. O sistema permite que o cliente compre vários trechos de viagens e reserve o hotel. E se o cliente conseguir comprar um trecho e não conseguir reservar o hotel? Ele vai viajar sem ter um hotel reservado? Nesse cenário, a compra das passagens aéreas e a reserva do hotel fazem parte da mesma transação; se uma etapa ou operação falhar, toda a transação falha.

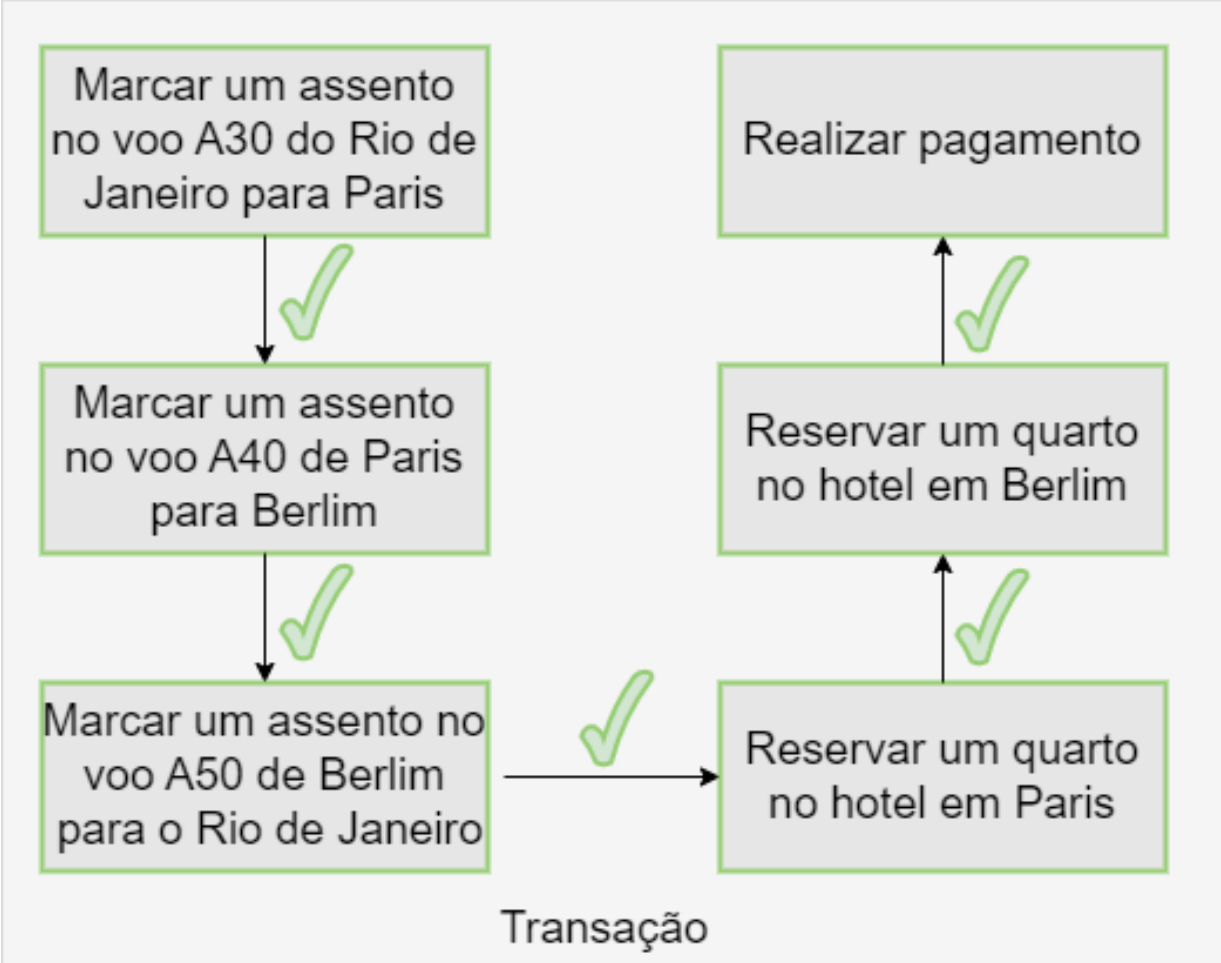

Figura 3.26: Sequência de operações de uma transação de reserva de passagem aérea e hotel.

No modelo centralizado, quando utilizamos sistemas monolíticos com um único banco de dados para todos os módulos, garantir a consistência do dado é mais simples, pois todas as tabelas de cada módulo residem no mesmo banco de dados. Usamos de forma transparente a propriedade ACID (*atomicidade, consistência, isolamento* e *durabilidade*) (https://pt.wikipedia.org/wiki/ACID) dos bancos de dados, que garante que todas as operações sejam tratadas como uma única transação — ou toda a transação é executada com sucesso, ou ela toda falha.

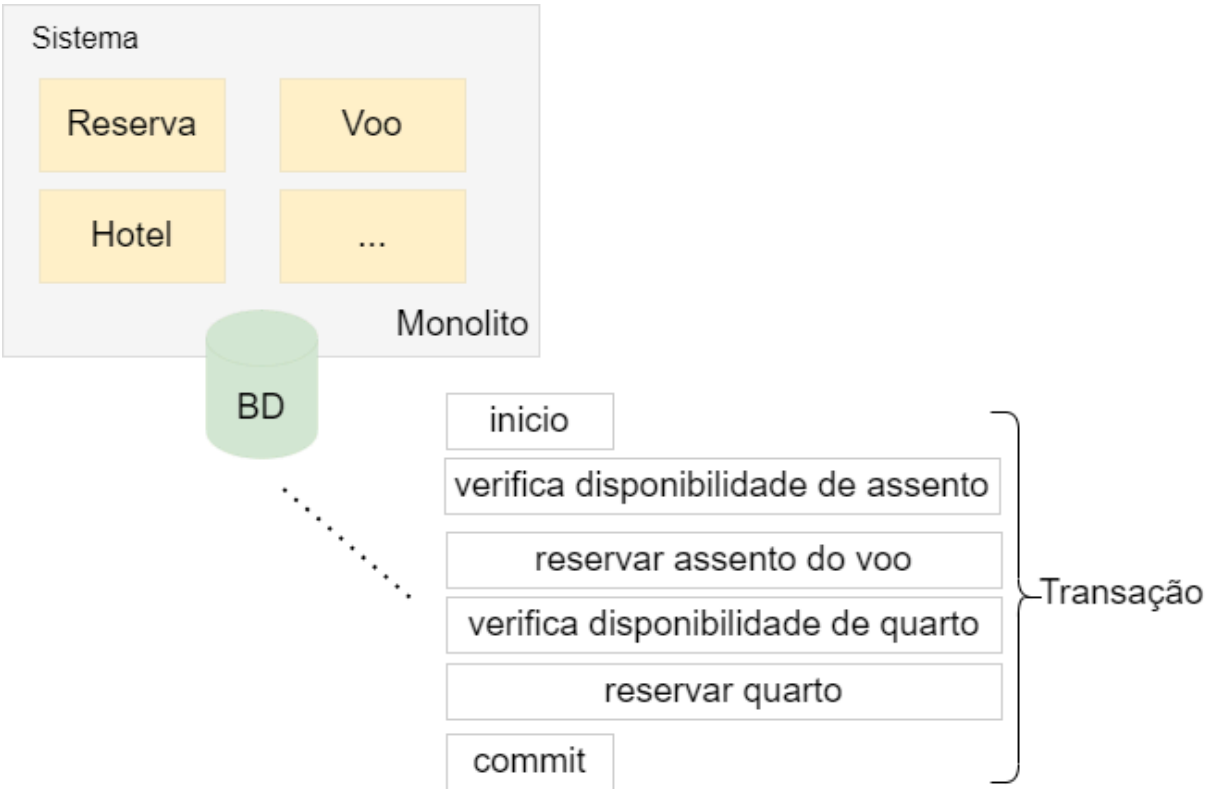

Figura 3.27: Sistema monolito com um único banco de dados ACID.

Na figura anterior, o banco de dados garante a consistência da transação. Caso a operação de reservar um quarto falhe, o *rollback* é realizado e a reserva do assento do voo não é efetivada/comitada.

Voltando para o nosso mundo distribuído, como garantimos a consistência do dado através de uma transação que abrange múltiplos serviços, cada um com seu banco de dados? O padrão arquitetural **SAGA** (ou seu plural *SAGAS*) é a forma como implementamos uma transação que abrange múltiplos serviços.

SAGA é uma sequência de transações locais, onde cada transação local tem sua compensação correspondente para desfazer suas alterações. Cada transação local atualiza seu dado dentro do seu serviço/banco de dados usando a propriedade ACID.

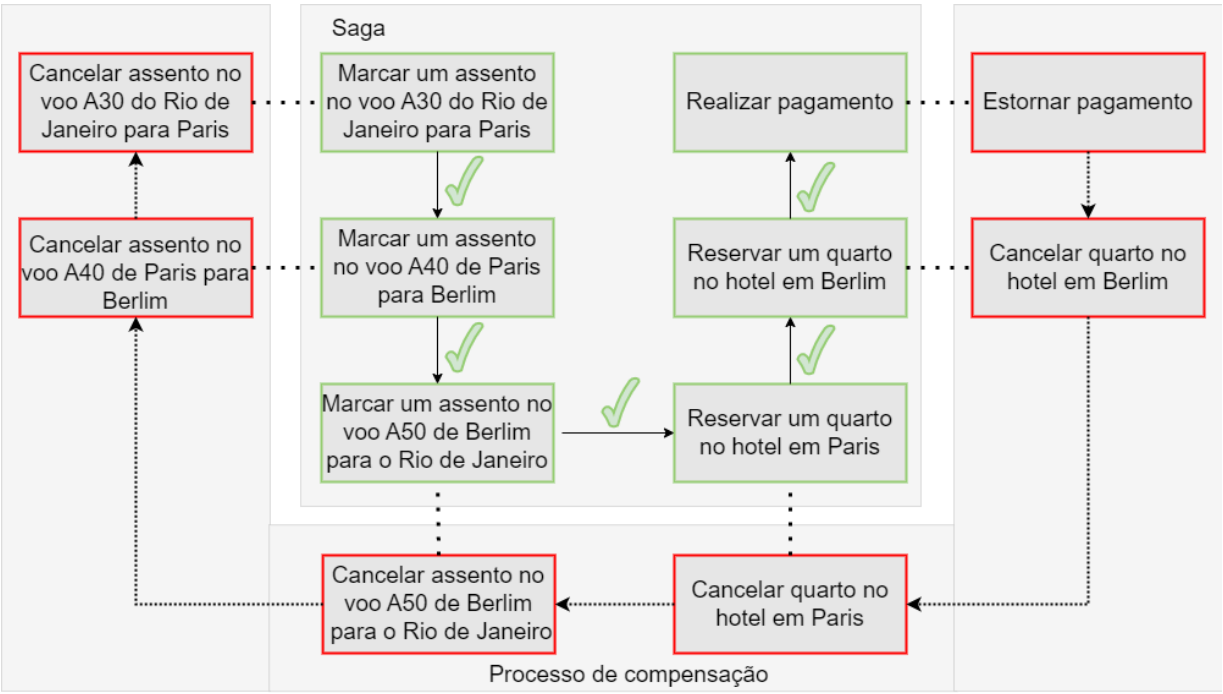

Figura 3.28: Sequência de operações de uma transação de reserva de passagem aérea e hotel com sua sequência de compensação.

Cada etapa ou operação dentro do SAGA abrange vários serviços. Esses serviços são conhecidos como participantes do SAGA. O fluxo de compensação é iniciado a partir da compensação correspondente da operação do participante do SAGA que falhou. Por exemplo, se a operação "*Reservar um quarto no hotel em Paris*" falhar, o fluxo de compensação é iniciado a partir do "*Cancelar quarto no hotel em Paris*" até a compensação correspondente da primeira operação, desfazendo todas as alterações realizadas.

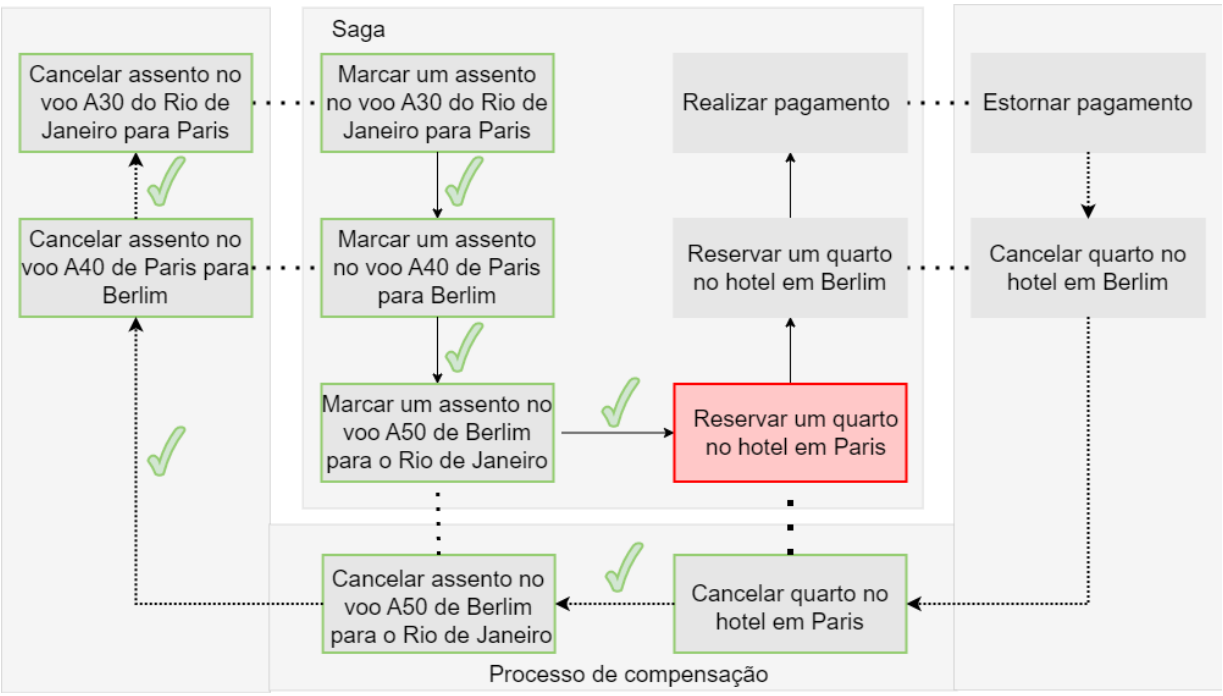

Figura 3.29: Processo de compensação em ação a partir de uma falha na operação do SAGA.

Tanto o SAGA quanto o fluxo de compensação é modelado de acordo com o processo de negócio. Pode ocorrer de haver dentro do processo operações de consultas onde não haja uma compensação correspondente.

Temos duas variações na forma de implementar o SAGA: com base na *orquestração* e na *coreografia*.

SAGA orquestrado

O SAGA orquestrado tem um controlador centralizado que rege toda a execução dos participantes do SAGA.

Para executar uma etapa do SAGA, ele envia uma mensagem de comando a um participante informando qual operação realizar. Após o participante do SAGA realizar a operação, ele envia uma mensagem de resposta ao orquestrador. O orquestrador então processa a mensagem e determina qual etapa do SAGA executar a

seguir. O orquestrador também lida com o processo de compensação.

Podemos utilizar o padrão de comunicação *request-reply*, mas nada impede de a comunicação ser síncrona com *request-response*. A ideia é ter o orquestrador controlando a sequência da execução dos participantes.

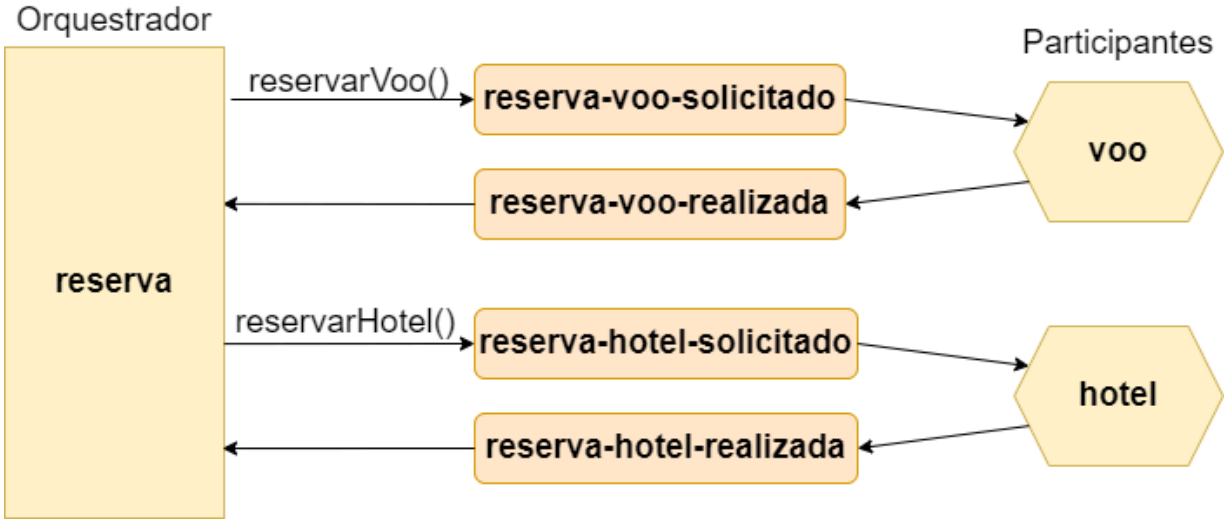

Figura 3.30: SAGA orquestrado para reserva de voo e hotel.

Uma de suas grandes vantagens é que ele dá uma visão clara da sequência das etapas do SAGA. Fica mais fácil entender o processo. Sua desvantagem é que o orquestrador acaba sendo um ponto de falha por gerenciar todo o processo, além da complexidade que adiciona na solução por implementar toda uma lógica de coordenação de execução dos participantes do SAGA.

Por dar uma visão melhor da sequência das etapas do SAGA, essa variação é a mais indicada quando temos um processo complexo, por ter um controle centralizado dos participantes. Essa complexidade refere-se à quantidade de participantes envolvidos juntamente à sequência de execução necessária para atingir o objetivo do negócio, incluindo o fluxo de compensação.

Para mitigar o problema de ponto de falha e simplificar a implementação, você não precisa desenvolver um orquestrador do zero. Podemos usar serviços gerenciados como AWS Step Functions (https://aws.amazon.com/pt/step-functions/) ou Azure Durable Functions (https://learn.microsoft.com/en-us/azure/azure-functions/durable/).

SAGA coreografado

O SAGA coreografado é onde todos os participantes trocam eventos sem um ponto de controle centralizado. Para executar uma etapa do SAGA, o participante executa sua transação local e em seguida publica um evento notificando que uma mudança de estado ocorreu. Outros participantes consomem esse evento e seguem com a coreografia de eventos até atingir o objetivo do negócio. O fluxo de compensação também é acionado por eventos.

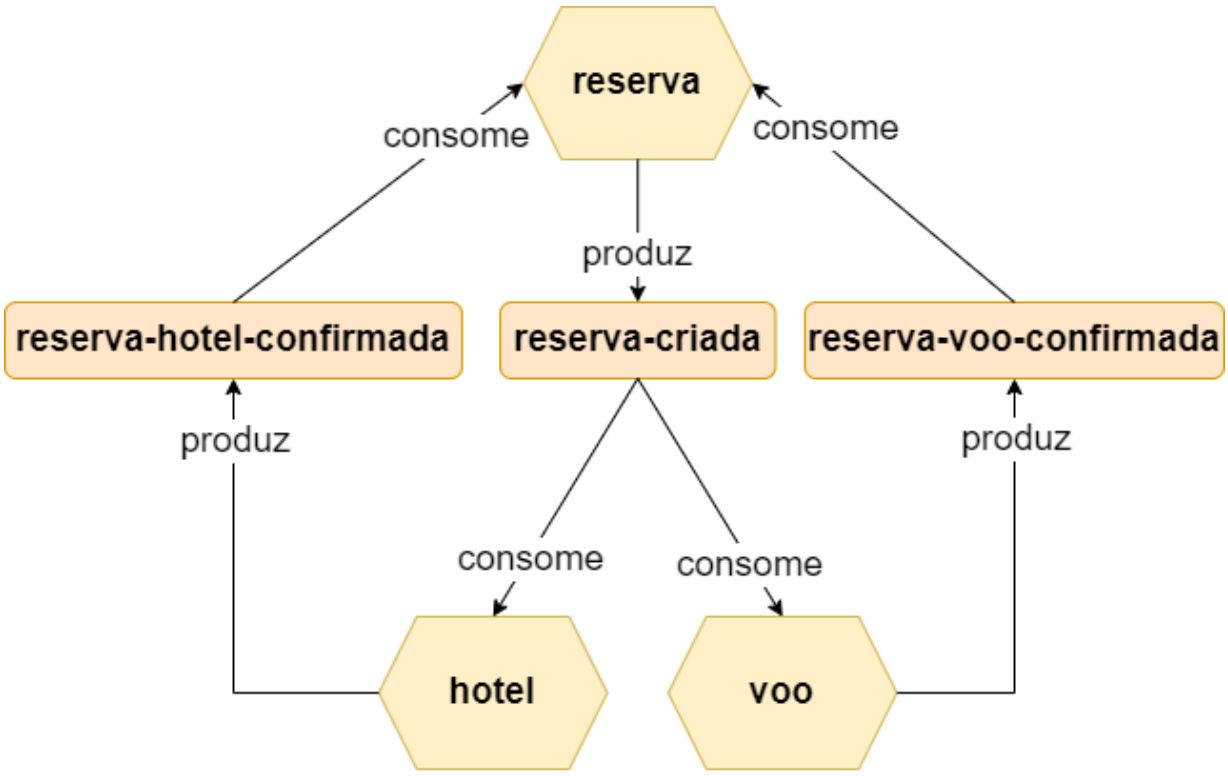

Figura 3.31: SAGA coreografado para reserva de voo e hotel.

Uma de suas grandes vantagens é sua natureza desacoplada, que se encaixa perfeitamente nos princípios de EDA, além de não requerer um orquestrador adicional para implementar e manter.

Sua desvantagem é que o SAGA pode se tornar confuso dependendo da quantidade de participantes, pois é difícil rastreá-los na coreografia dos eventos. Realizar teste também se torna mais desafiador, por não estar claro o fluxo de execução na coreografia.

Essa variação é ideal para processos simples, que requerem poucos participantes e não precisam de uma lógica para coordená-los. Como uma solução pode contemplar vários processos de negócio, dependendo do caso de uso, nada impede de usarmos as duas variações na mesma solução EDA.

Vamos voltar ao nosso exemplo hipotético de e-commerce. Imagine que o processo de negócio para entregar um pedido de um cliente tenha os seguintes passos:

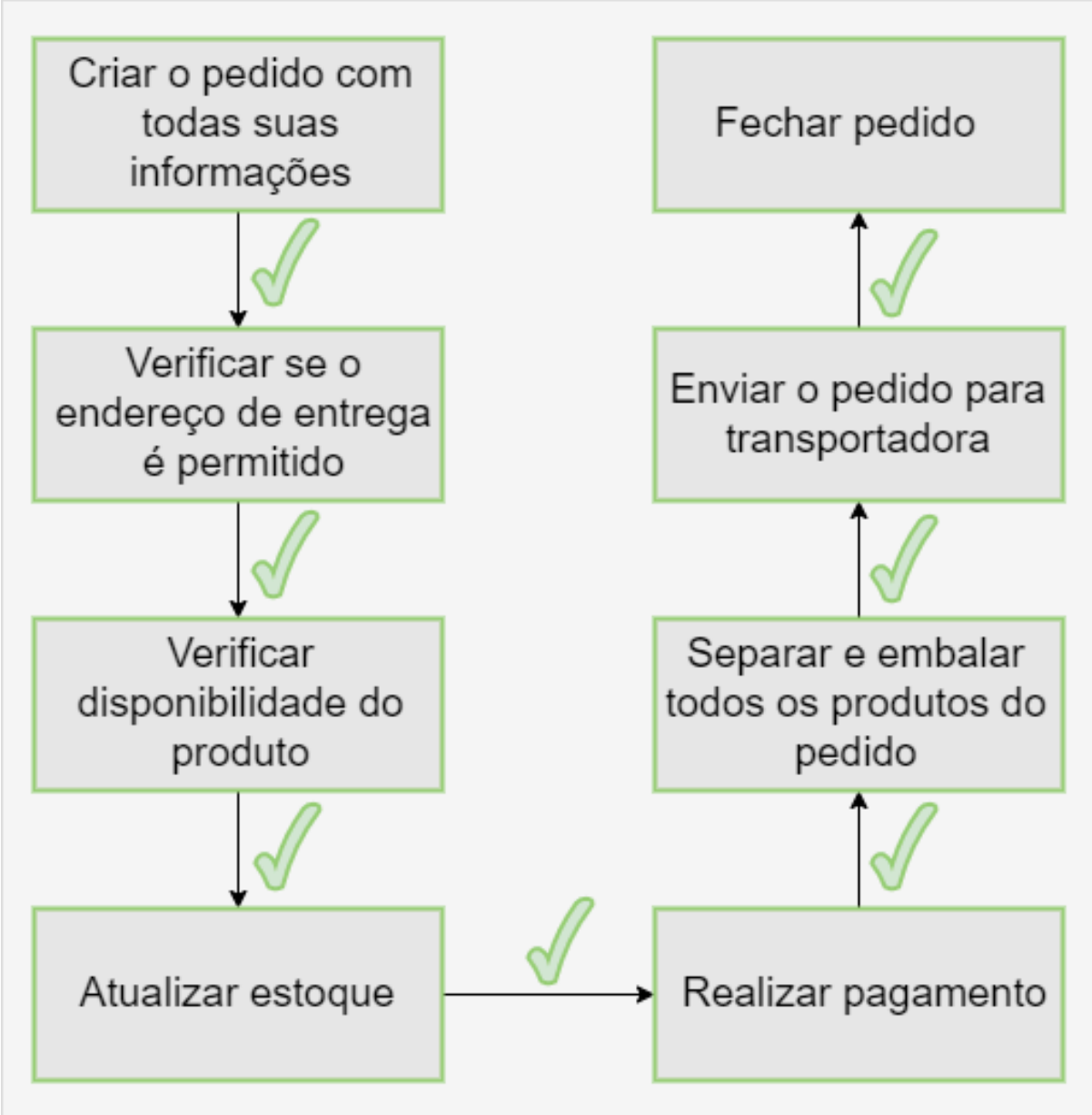

Figura 3.32: Processo de entrega no e-commerce.

Vamos projetar nossa solução com a coreografia de eventos para atender ao processo de entrega do e-commerce utilizando o SAGA coreografado. Temos várias opções de solução da nossa coreografia. Para ganharmos agilidade na entrega do produto, quando um pedido for criado, podemos paralelizar todo o processo, conforme a proposta de solução a seguir:

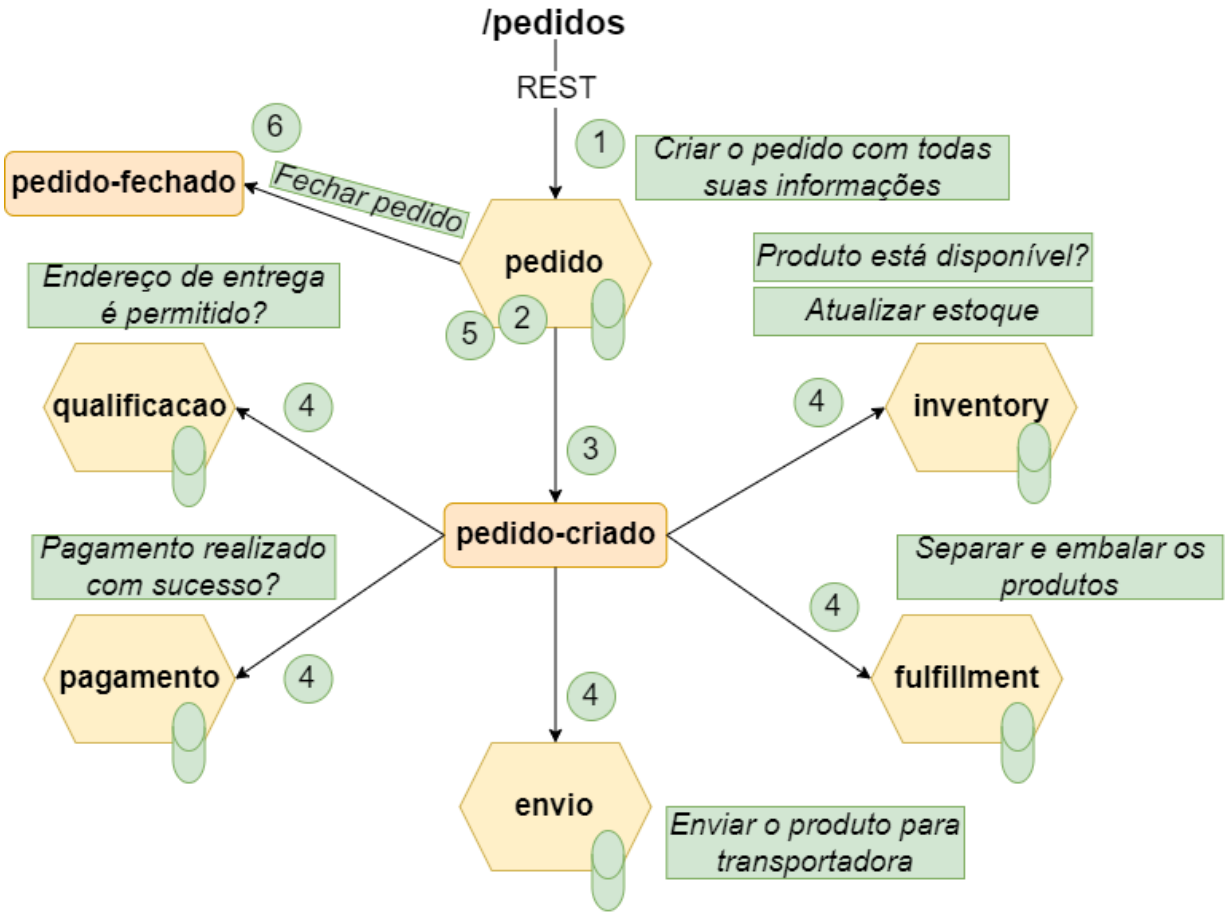

Figura 3.33: Proposta de solução de e-commerce paralelizando todos os passos do processo.

1. Cliente faz o seu pedido;
2. O microsserviço `pedido` cria o pedido com todas suas informações;
3. O microsserviço `pedido` publica o evento `pedido-criado`;
4. Os microsserviços `qualificacao`, `pagamento`, `inventory`, `fulfillment` e `envio` subscrevem o evento `pedido-criado` e executam suas tarefas em paralelo;
5. Para deixar o fluxo mais simples de visualizar, omiti os eventos correspondentes dos microsserviços `qualificacao`, `pagamento`, `inventory`, `fulfillment` e `envio` após a conclusão de suas tarefas. Todos esses eventos de conclusão podem ser consumidos pelo microsserviço `pedido` para atualizar a situação do pedido de acordo;

6. Após receber toda a conclusão, o pedido é fechado e, então, é publicado o evento `pedido-fechado`.

Essa abordagem pode nos trazer uma maior agilidade no processo, pois boa parte dele está sendo executada em paralelo. No caminho feliz do fluxo, pode até fazer sentido. Mas e se o endereço de entrega não for permitido? Ou o produto não está disponível? E se o pagamento não for autorizado? **A forma com que projetamos nossa coreografia pode adicionar uma grande complexidade na nossa solução**. Quando projetamos, temos que levar em consideração o fluxo de compensação.

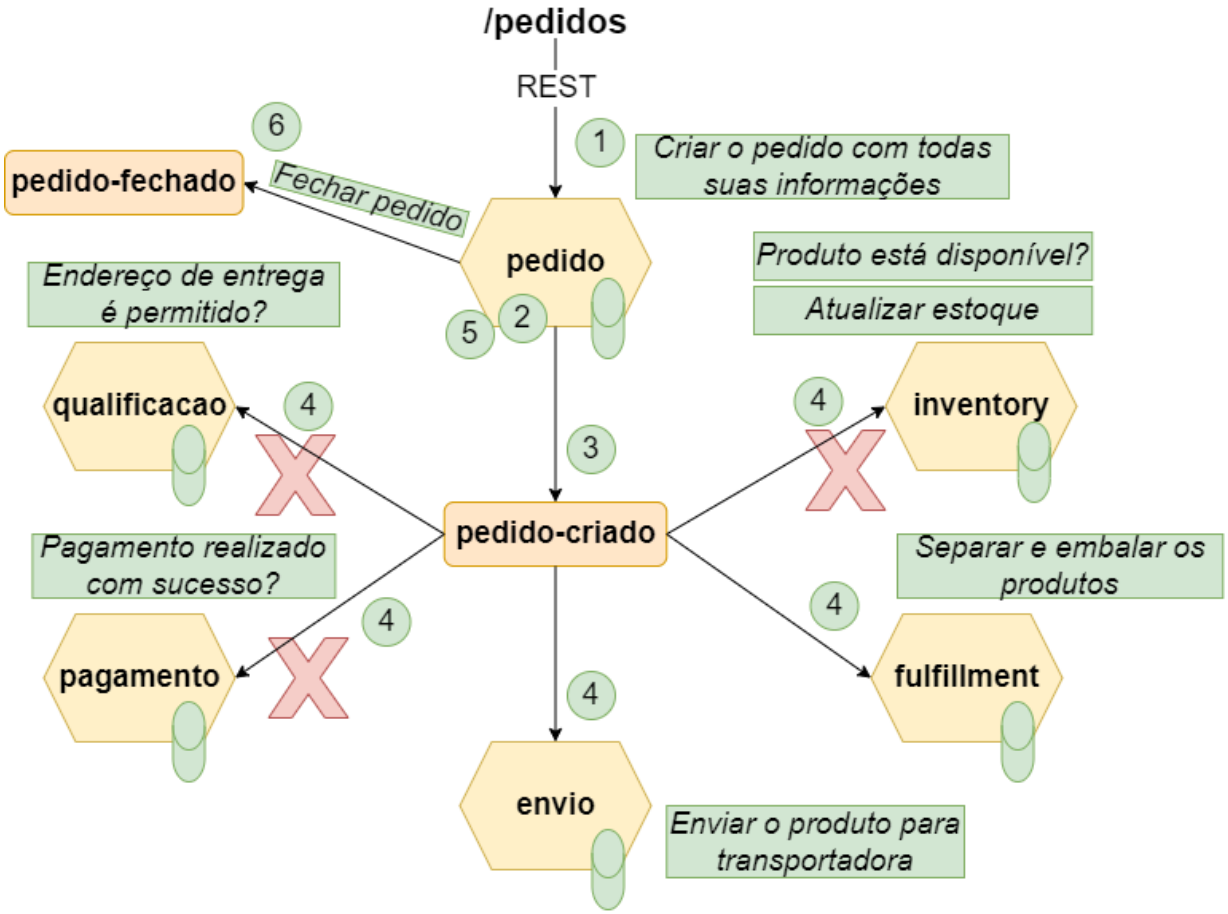

Figura 3.34: Proposta de solução de e-commerce demonstrando possíveis erros no fluxo.

Como podemos projetar nossa solução para mitigar as transações de compensação?

Primeiro, entender o fluxo de compensação desse processo. Esse fluxo é muito importante e pode nos direcionar em como projetamos nossa coreografia de eventos.

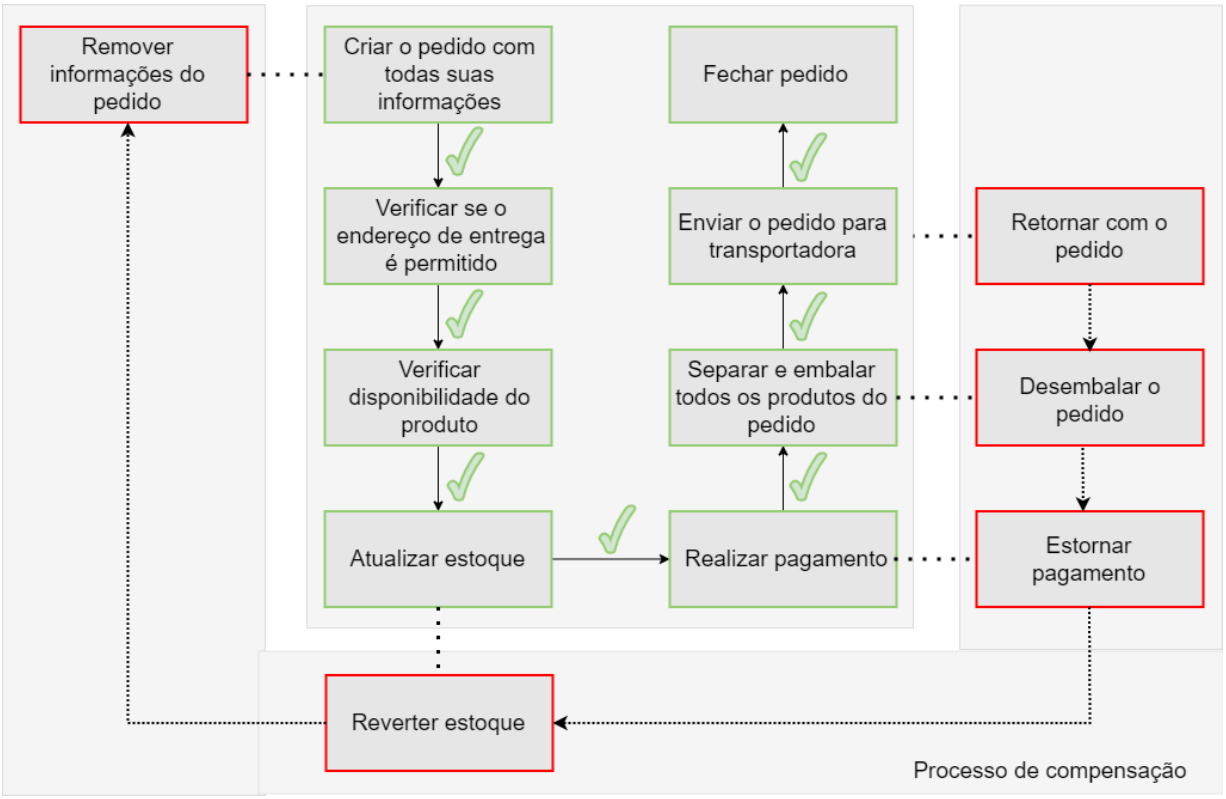

Figura 3.35: Processo de entrega no e-commerce e fluxo de compensação.

Estando claro o processo de entrega e o fluxo de compensação, começam os questionamentos: será que a agilidade do paralelismo do caminho feliz compensa o custo do fluxo de compensação? Qual a chance de o produto não ter em estoque? Será que vale a pena realizar o pagamento em paralelo? Qual a chance de o pagamento ser negado? Ocorre com frequência? Será que não seria menos custoso para a empresa iniciarmos a separação e o embalo do produto apenas depois de receber a confirmação do pagamento? Os passos de verificação de disponibilidade de endereço e verificação de estoque fazem sentido ser em paralelo? Se um falhar, será custoso para a empresa executar o fluxo de compensação ou é indiferente? Toda essa análise é crucial para a solução.

A seguir, temos uma segunda proposta de solução após uma análise mais minuciosa no processo de entrega e no fluxo de compensação desse nosso exemplo simplificado de e-commerce.

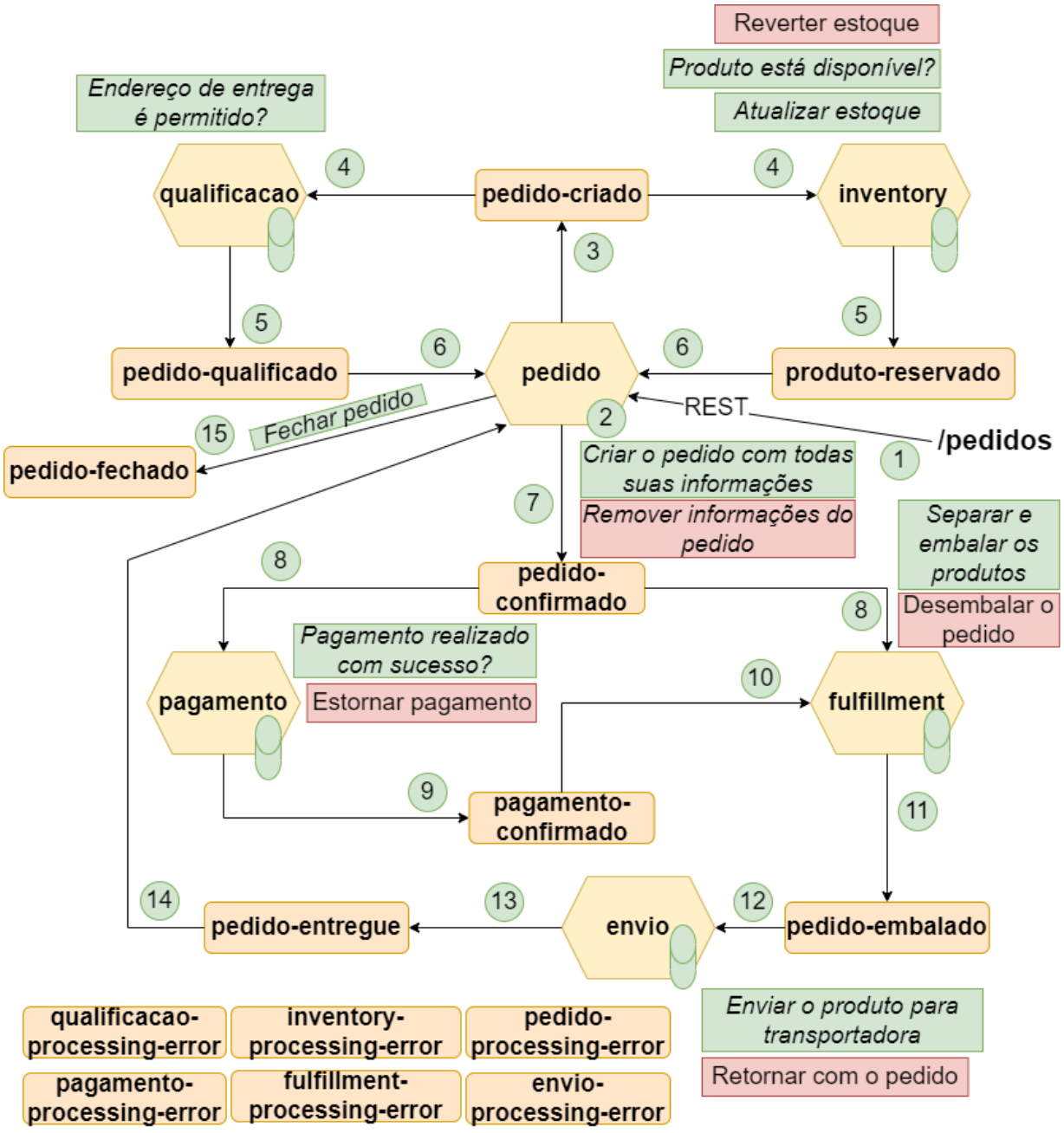

Figura 3.36: Coreografia de eventos do projeto de e-commerce simplificado.

1. Cliente faz o seu pedido;

2. O microsserviço `pedido` cria o pedido com todas suas informações;
3. O microsserviço `pedido` publica o evento `pedido-criado` ;
4. Os microsserviços `qualificacao` e `inventory` consomem o evento `pedido-criado` . O primeiro verifica a disponibilidade do endereço de entrega e o segundo verifica a disponibilidade do produto e atualiza o estoque;
5. Os microsserviços `qualificacao` e `inventory` publicam seus eventos de conclusão `pedido-qualificado` e `produto-reservado` , respectivamente;
6. O microsserviço `pedido` consome os eventos `pedido-qualificado` e `produto-reservado` e atualiza a situação do pedido;
7. O microsserviço `pedido` publica o evento `pedido-confirmado` ;
8. Os microsserviços `pagamento` e `fulfillment` consomem o evento `pedido-confirmado` para realizar o pagamento e iniciar o processo de separação e embalagem, respectivamente (Obs.: o negócio entende que vale a pena realizar o pagamento e já iniciar o processo de separação e embalagem em paralelo devido à porcentagem de pagamento negado, mas o processo de separação e embalagem só será finalizado após a confirmação do pagamento);
9. O microsserviço `pedido` atualiza a situação do pedido e publica o evento `pagamento-confirmado` ;
10. Uma vez que o pagamento foi confirmado, o processo de separação e embalagem é finalizado;
11. O microsserviço `fulfillment` publica o evento `pedido-embalado` ;
12. Com o pedido embalado e pronto para entrega, o microsserviço `envio` consome o evento `pedido-embalado` e inicia o processo de entrega com a transportadora (a comunicação com a transportadora foi omitida no desenho);
13. O microsserviço `envio` recebe a notificação do sistema de transporte e publica o evento `pedido-entregue` quando o pedido for entregue no endereço do cliente;
14. O microsserviço `pedido` consome o evento `pedido-entregue` ;
15. O microsserviço `pedido` finaliza o pedido em sua base de dados e publica o evento `pedido-fechado` .

Cada microsserviço da solução pode publicar um evento de erro genérico. Esses eventos são o gatilho do fluxo de compensação. O motivo do erro fica no corpo do evento. Deixei os erros genéricos ao lado da coreografia para simplificar a visualização do desenho.

Imagine que temos cinco aplicações consumidoras que precisam ser notificadas quando houver um pagamento negado. Seria eficiente, a cada erro genérico publicado pelo microsserviço de pagamento, esses consumidores verificarem o conteúdo do evento para saber se o motivo foi de pagamento negado? Obviamente não: isso poderia gerar processamento desnecessário nessas aplicações, pois nem todo erro seria de pagamento negado.

Provavelmente, no mundo real, além desses eventos de erros genéricos, podemos ter outros mais específicos que também podem ser um gatilho do fluxo de compensação e de outros eventos de negócio — por exemplo, `PagamentoNegado`, `ProdutoIndisponivel` e por aí vai. Nesse caso, essas aplicações consumidoras seriam notificadas apenas quando o evento `PagamentoNegado` ocorresse, sem precisar fazer uma verificação extra. Esse é um exemplo de que o evento de erro genérico não é ideal para esses casos.

A seguir, temos três exemplos de desenho para parte do fluxo de compensação.

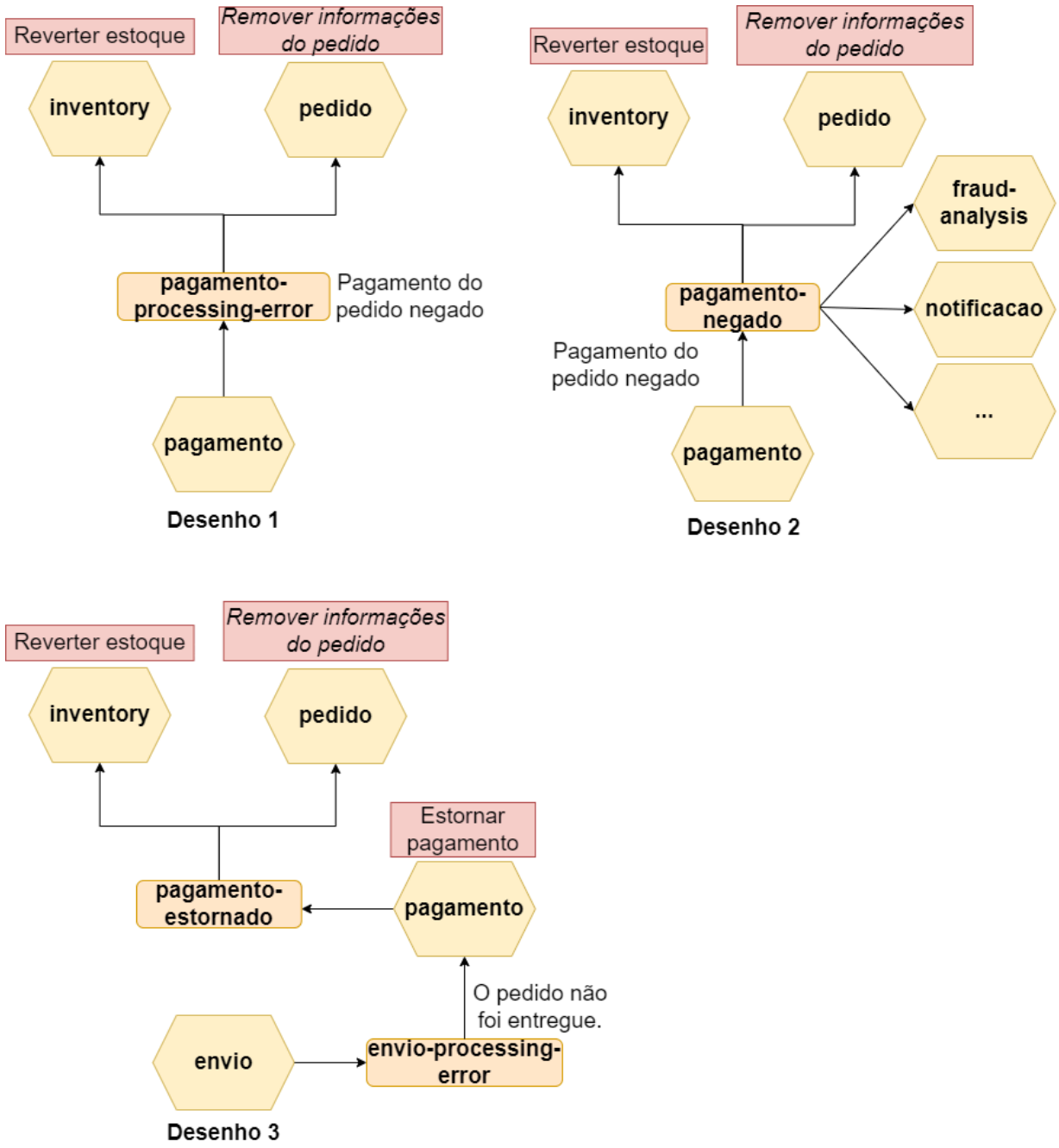

Figura 3.37: Coreografia de eventos do fluxo de compensação.

Desenho 1

O microsserviço `pagamento` publica um evento de erro genérico informando no seu corpo que o pagamento foi negado. Isso é um gatilho para os microsserviços `inventory` e `pedido` realizarem suas compensações. Eles executam as compensações em paralelo.

Desenho 2

O microsserviço `pagamento` publica um evento de erro específico de *pagamento negado*. Isso é um gatilho para os microsserviços `inventory` e `pedido` realizarem suas compensações. Além disso, é um gatilho para outros microsserviços executarem suas tarefas seguindo a coreografia de eventos.

Desenho 3

Na estratégia do desenho 3, quando o pedido, por algum motivo, não for entregue no endereço do cliente, primeiro o fluxo de compensação estorna o pagamento. Somente após a confirmação do estorno do pagamento, os microsserviços `inventory` e `pedido` realizam suas compensações.

Em todo projeto de EDA temos como resultado o desenho da coreografia. Nos projetos de que participo, chamo-os de "*The Big Picture*". Mas será que apenas o processo com os passos do fluxo de entrega e compensação é o suficiente para projetarmos a coreografia de eventos? A resposta rápida é "não". Com o processo definido, executamos um workshop chamado ***EventStorming*** para levantarmos, dentre outras coisas, os eventos possíveis dentro do processo. Falaremos sobre *EventStorming* no próximo capítulo, *Identificando eventos de negócio e documentando nossa solução*.

Também podemos ter diagramas adicionais para nos ajudar a comunicar melhor nossa solução. Apesar de a coreografia de eventos não ser exatamente uma sequência, acho que vale a pena termos um diagrama de sequência para comunicar o fluxo do caminho feliz.

Para uma melhor visualização do diagrama de sequência, disponibilizei o arquivo `.drawio` no nosso projeto do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap3).

O padrão SAGA é peça-chave ao projetarmos nossa solução de EDA. Para uma solução bem-sucedida, o processo de negócio do cliente

(incluindo o fluxo de compensação) tem que estar claro desde o início do projeto. O SAGA atua exatamente em cima do processo de negócio como uma sequência de transações locais para garantir a consistência do fluxo. Pela minha experiência, é muito comum iniciarmos um novo projeto no cliente e seu processo de negócio não estar definido ou completo. Temos que ajudá-los a definir e, a partir disso, trabalhar em cima dele com o *EventStorming*.

Como vimos, o padrão SAGA é a forma que endereçamos o problema da consistência do dado através de uma transação que abrange múltiplos serviços. Mas, para cada transação local, precisamos comunicar, através de eventos, para os outros participantes da coreografia que a entidade de negócio teve alteração.

O problema da inconsistência ao salvar no banco de dados e publicar um evento

As aplicações ou microsserviços dentro da nossa coreografia de eventos executam operações de escrita (criação, atualização e remoção) em sua base de dados e, em seguida, publicam um evento notificando que uma alteração de estado de uma entidade de negócio ocorreu. Essa forma é a mais comum nos projetos de EDA.

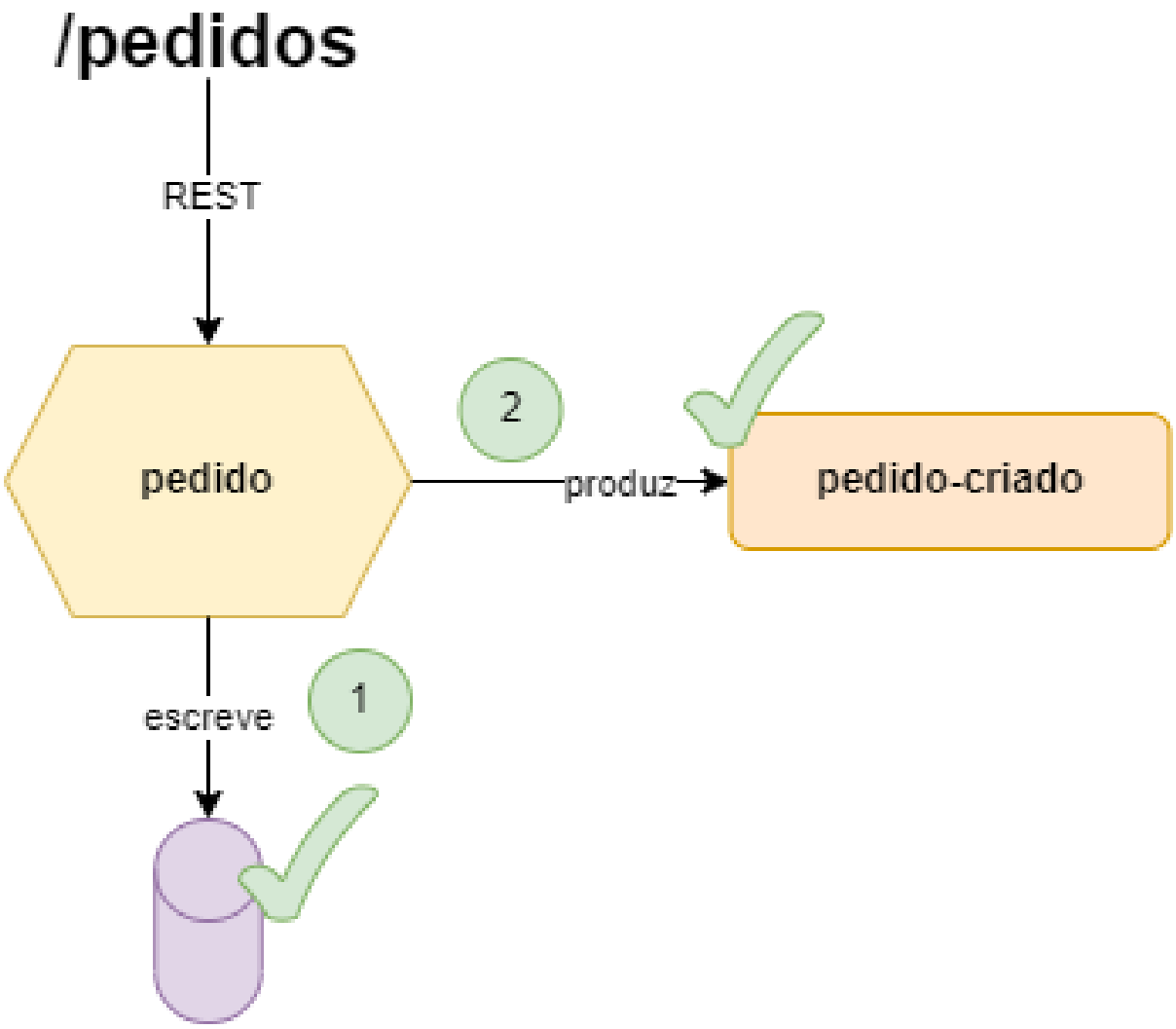

Figura 3.38: Operações de escrita no banco de dados e publicação de evento.

1. O microsserviço `pedido` cria o pedido com todas suas informações;
2. O microsserviço `pedido` publica o evento `pedido-criado`.

Esses dois passos ou operações (escrita no banco de dados e publicação do evento) não são atômicos, eles não fazem parte da mesma transação. E se a publicação do evento falhar? Outras aplicações não ficarão sabendo da mudança do estado dessa entidade de negócio.

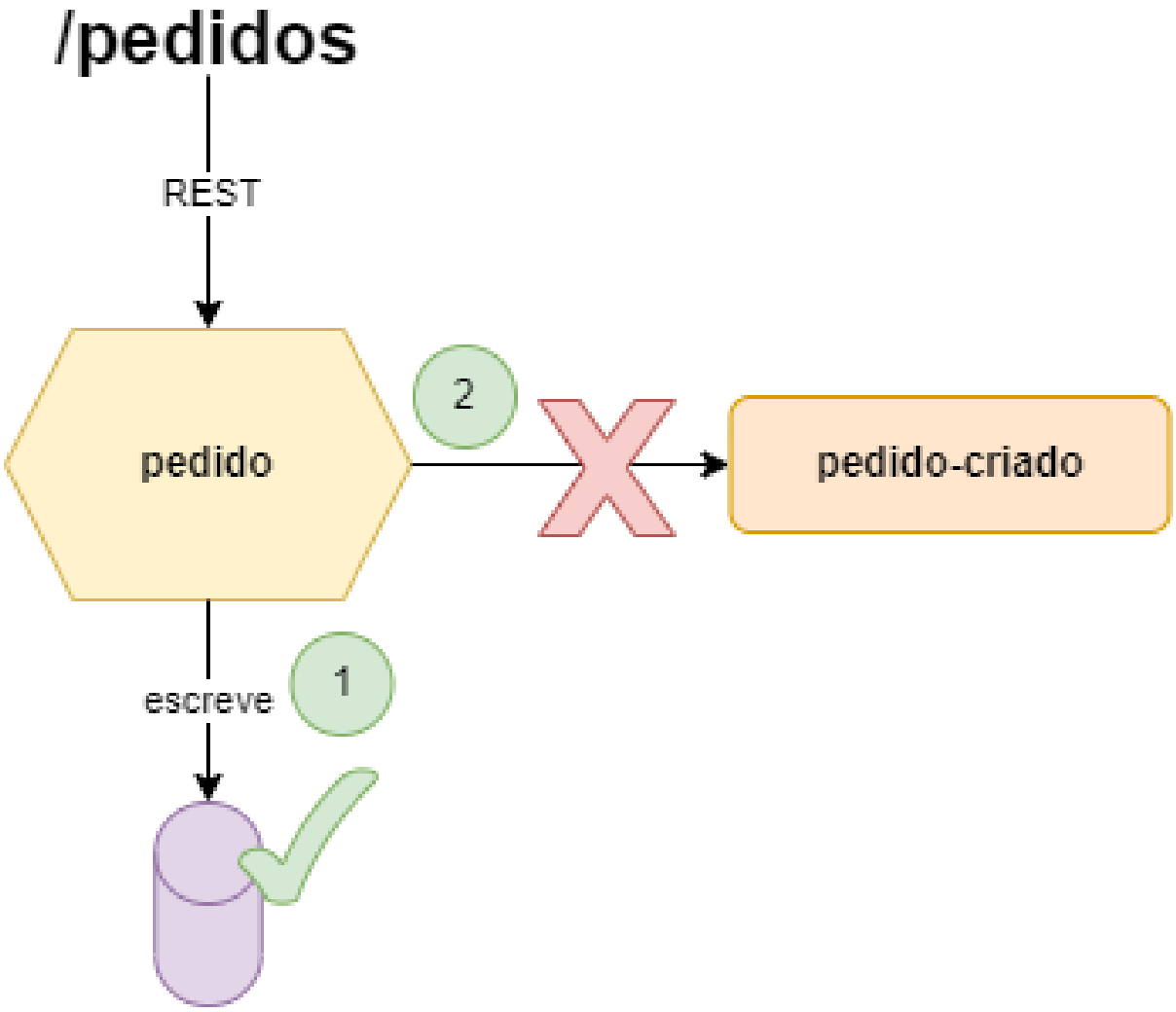

Figura 3.39: Operações de escrita no banco de dados e publicação de evento não realizado

1. O microsserviço `pedido` cria o pedido com todas suas informações;
2. O microsserviço `pedido` não consegue publicar o evento `pedido-criado` mesmo após retentativas;

A consistência do dado é garantida no banco de dados pela propriedade ACID, mas não no envio do evento para o *broker*. Mesmo com as possíveis retentativas para mitigar o não envio de eventos ao *broker*, ainda assim o *broker* pode não ser capaz de receber o evento.

Para garantirmos a consistência das duas operações (escrita no banco de dados e publicação do evento), podemos usar alguns padrões: os padrões *Transactional outbox* e o *Change Data Capture* (CDC).

3.7 Transactional outbox

O padrão *outbox* tem uma tabela adicional à tabela principal para armazenar o evento. Essas duas tabelas fazem parte do mesmo banco de dados, portanto uma operação de alteração nessas duas tabelas é atômica, faz parte da mesma transação.

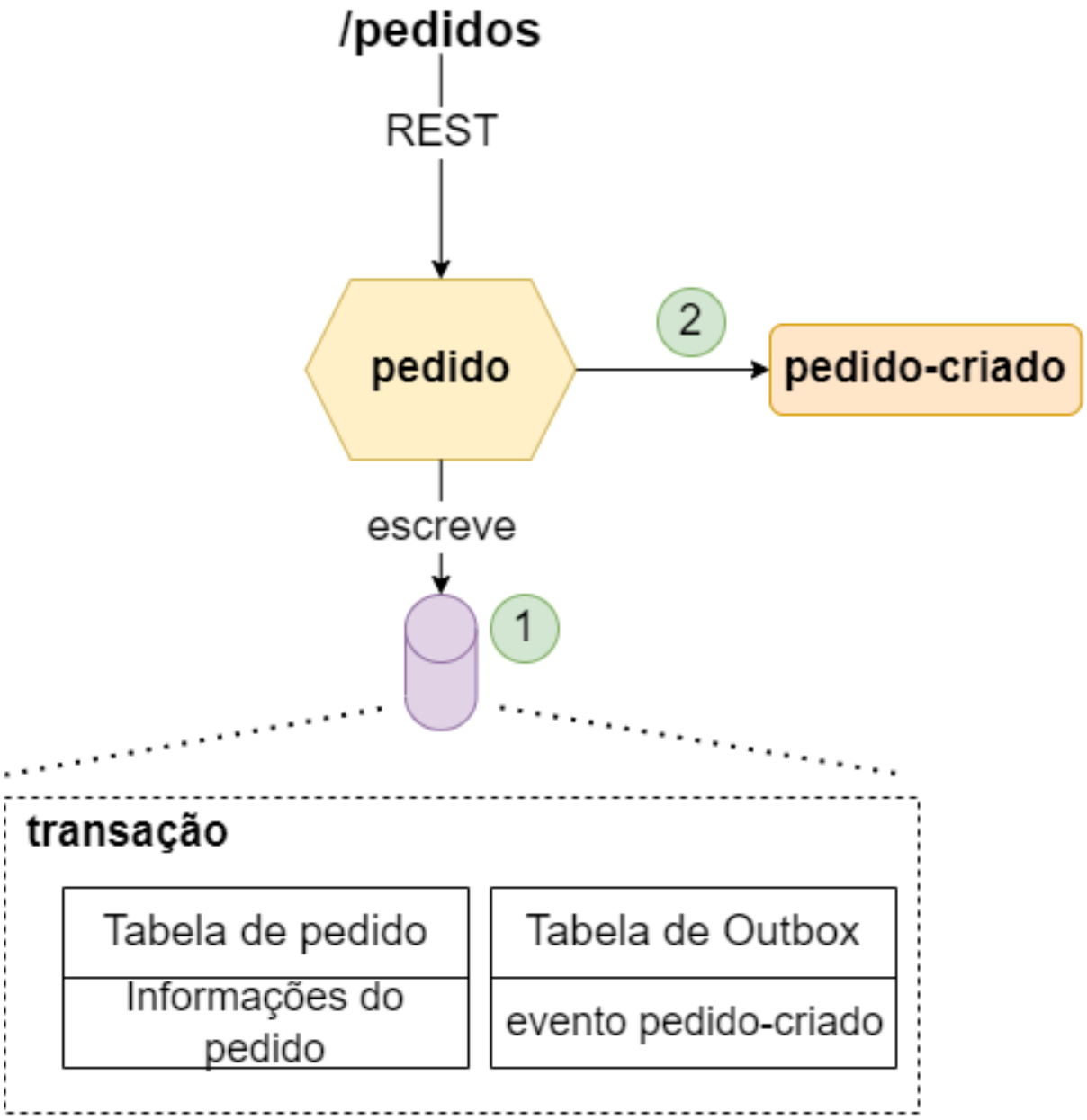

Figura 3.40: Padrão Transactional outbox.

Após a publicação do evento, o microsserviço `pedido` remove o registro da tabela outbox. Caso a publicação do evento falhe, o evento continuará na tabela, podendo ser posteriormente publicado.

Pode-se criar mecanismos para tentar publicar esses eventos que ficaram na tabela *outbox*. Por exemplo, ao iniciar o microsserviço,

ele verifica se tem eventos nessa tabela ou define processo de *polling*, entre outras opções.

3.8 Change data capture (CDC)

O CDC é um padrão que identifica e captura alterações no dado em tempo real, tipicamente em um banco de dados. Uma vez que um registro é alterado (criado, atualizado ou removido), ele publica um evento notificando a mudança no registro. Uma forma sofisticada de implementarmos o padrão CDC é utilizando um outro padrão, o *Transaction log tailing*.

Transaction log tailing

O servidor de banco de dados mantém um registro de cada alteração feita nas tabelas durante uma transação. Esse registro é o *transaction log* ou *commit log*. O *transaction log* é um componente crítico de um banco de dados. Se ocorrer qualquer falha no servidor, o banco de dados pode recorrer ao *transaction log* para trazer os dados de volta e com isso ficar em um estado consistente. Além disso, pode ser utilizado para operações de *rollback*.

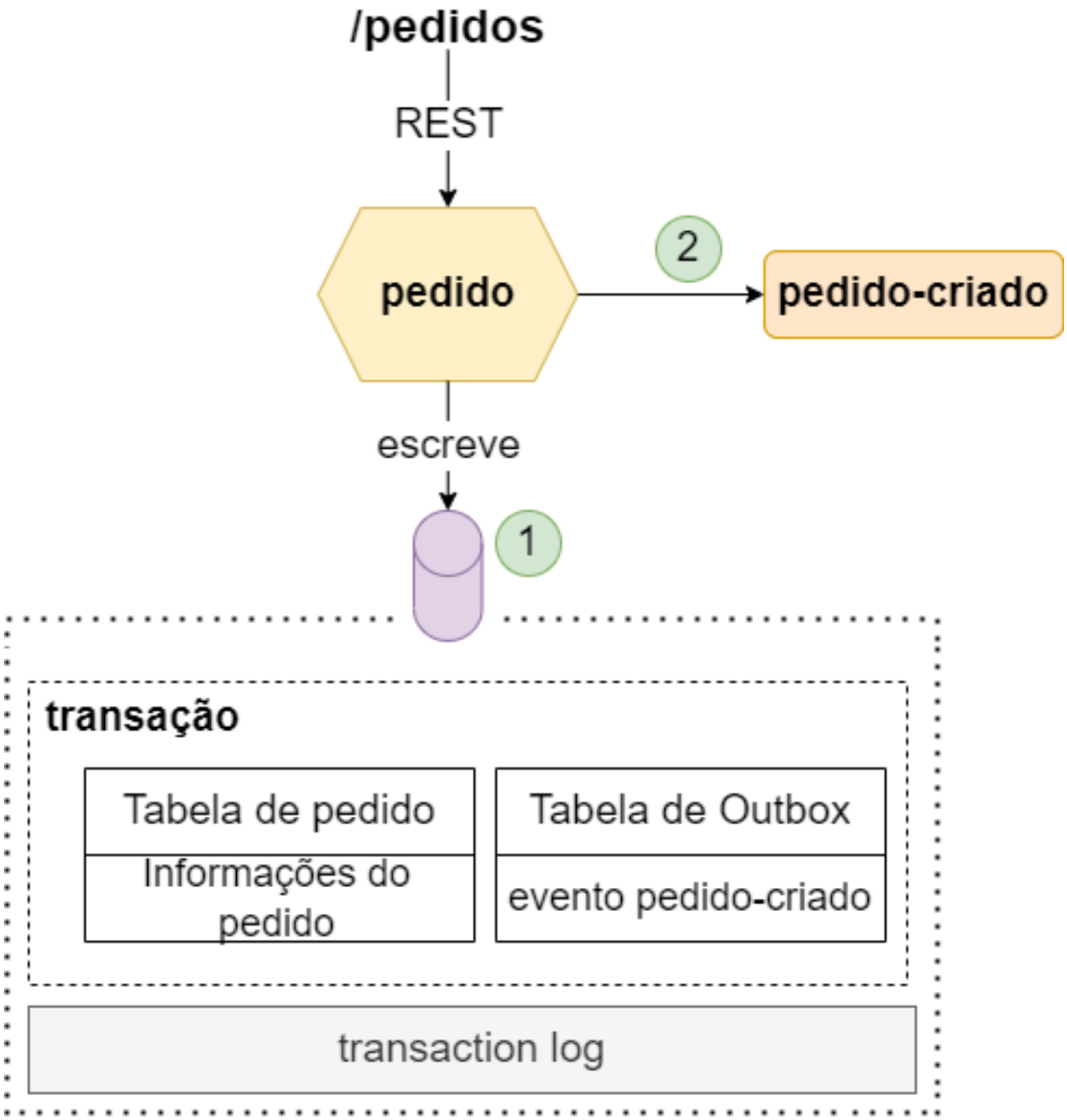

Figura 3.41: Padrão Transaction log tailing.

O padrão *transaction log tailing* é o padrão que permite capturar as alterações do banco de dados pelo *transaction log*.

Alguns bancos de dados disponibilizam funcionalidades nativas que suportam CDC para extrair as alterações, em nível de tabela, em seus respectivos *transaction logs*. Alguns exemplos são:

- Oracle LogMiner:

    - https://docs.oracle.com/en/database/oracle/oracle-database/19/sutil/oracle-logminer-utility.html
- MySQL binlog:
    - https://dev.mysql.com/doc/refman/8.0/en/binary-log.html
- AWS DynamoDB streams:
    - https://docs.aws.amazon.com/pt_br/amazondynamodb/latest/developerguide/streamsmain.html
- SQL Server CDC:
    - https://learn.microsoft.com/pt-br/sql/relational-databases/track-changes/about-change-data-capture-sql-server?view=sql-server-ver16
- PostgreSQL write-ahead log (WAL):
    - https://www.postgresql.org/docs/current/wal-intro.html

A forma nativa de extração disponibilizada pelos fornecedores dos bancos de dados possibilita a extração e não a publicação do evento. Fica a cargo da solução desenvolver o componente que consulta o dado alterado e publica o evento.

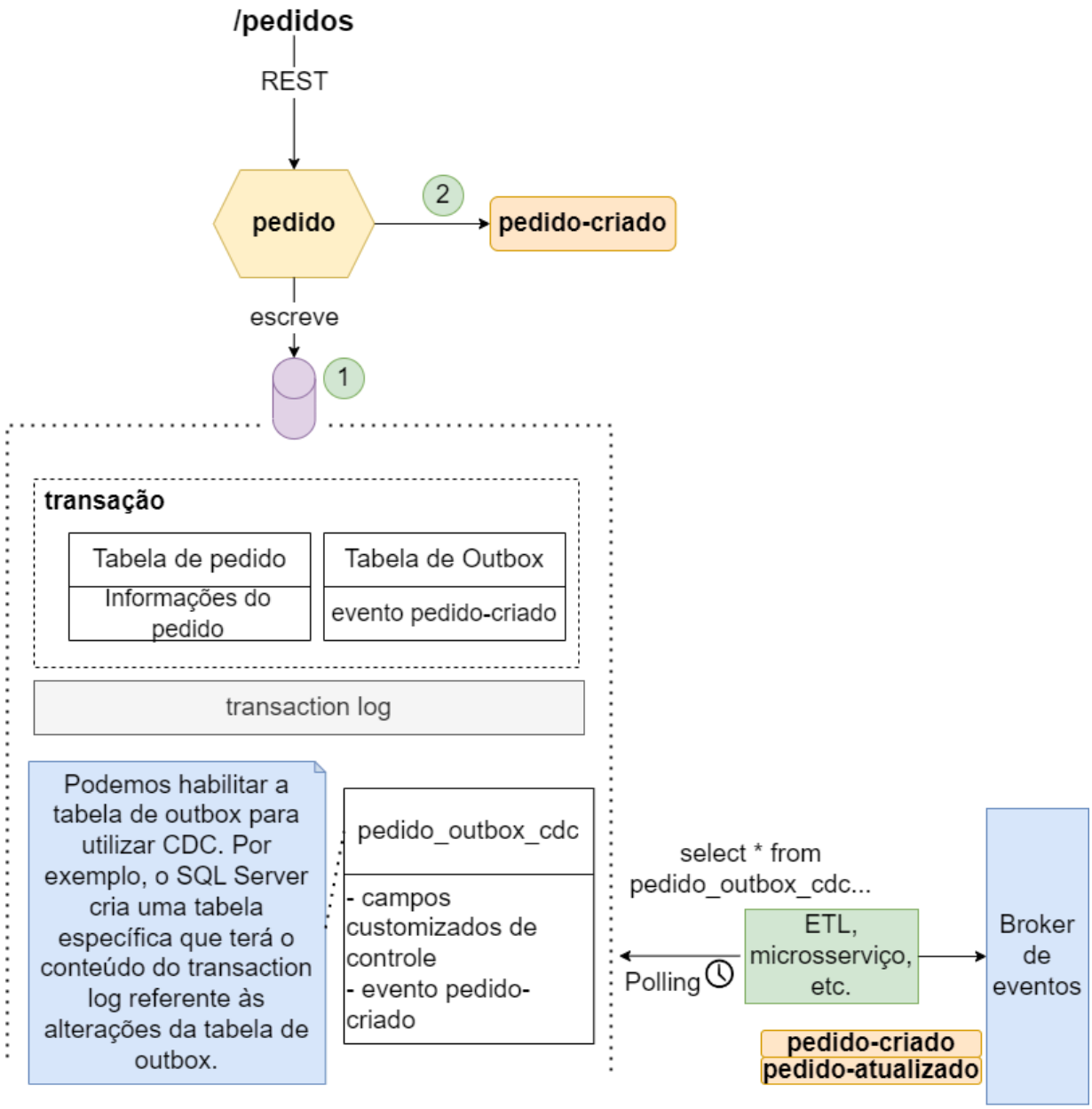

Figura 3.42: Exemplo de CDC utilizando com componente customizado para publicação de eventos.

A grande desvantagem de fazermos *polling* no banco de dados é que, dependendo do intervalo configurado, o tempo entre o dado efetivado/comitado no banco e a publicação do evento pode impactar o negócio, não usufruindo do processamento em tempo real da EDA. Também há a possibilidade de consumo desnecessário

de recursos no banco de dados, já que nem sempre virão dados na resposta da *query*.

Com isso, surgiram plataformas e frameworks que abstraem toda essa complexidade de captura e publicação de eventos através de conectores, que necessitam apenas de algumas configurações, como o Debezium (https://debezium.io/) e o Kafka Connect, da Confluent (https://docs.confluent.io/platform/current/connect/index.html#kafka-connect).

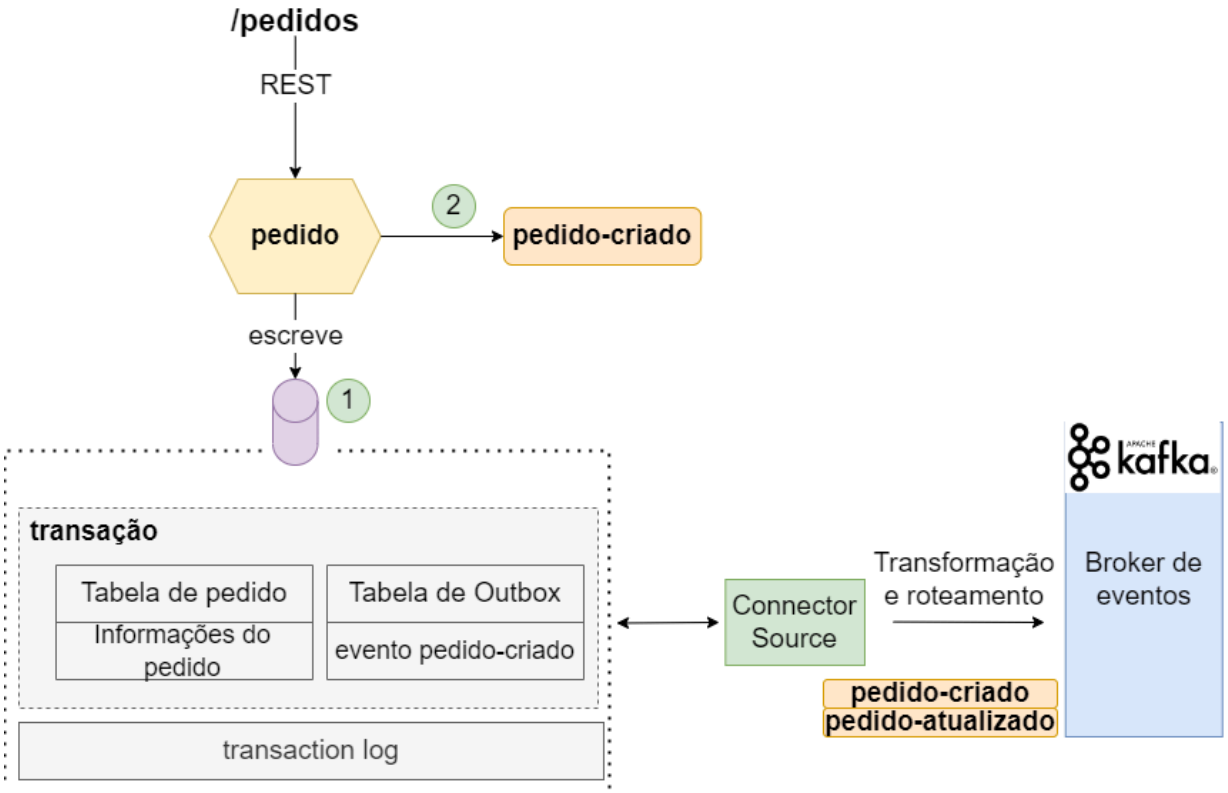

Figura 3.43: Exemplo de CDC utilizando uma plataforma com conectores.

A forma como o conector se integra com o banco de dados fica transparente na nossa solução. Podemos utilizar a captura diretamente na tabela de pedido ou na tabela de *outbox*. A vantagem de utilizarmos a tabela de *outbox* é que não precisaríamos ter um processo de transformação de dados do esquema do banco de dados para o esquema do evento, a menos que estivéssemos utilizando o padrão *event sourcing*.

Um caso de uso bem comum em que utilizamos o padrão CDC é na migração de um sistema monolítico para uma arquitetura de microsserviço e EDA.

Hugo Rocha, em seu livro *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices* (2021), conta sua experiência em não usar o padrão CDC em um projeto de migração e o problema que ele enfrentou em produção por causa das alterações nos dados "camufladas" dentro de *stored procedures*.

Apesar de podermos usar esses padrões para resolver o problema da inconsistência, não costumo utilizá-los no fluxo do caminho feliz da coreografia dos eventos em todos os projetos de EDA, para não adicionar uma maior complexidade na solução. Por termos soluções de banco de dados e *broker* de eventos com alta disponibilidade, e ainda mitigado pelas retentativas nos casos de erros transitórios, normalmente prefiro definir um processo específico para os casos de erros (não conseguir publicar o evento). Seja um processo manual (por exemplo, o time de operações publicando o evento manualmente) ou, de preferência, automatizado (por exemplo, a monitoração iniciar o processo de retentativa de envio do evento após um *health check* bem-sucedido no *broker*), as possibilidades de solução para esses casos de erros são diversas.

Muitos projetos de EDA falham por não garantirmos a consistência dessas operações (escrita no banco de dados e publicação do evento), seja por não implementarmos os padrões vistos anteriormente ou por não definirmos um processo específico para esses casos de erros.

3.9 Webhooks

Webhooks são notificações *push* baseadas em `HTTP(S)` acionadas quando um evento ocorre. Eles permitem que uma aplicação saiba que um evento ocorreu no sistema de origem. Muito utilizado na integração de sistemas de forma geral, *webhooks* são conhecidos como API de notificação ou API de eventos no contexto do sistema de origem.

O cliente registra seu *endpoint* `HTTP(S)` no sistema origem e habilita sobre qual evento ele quer ser notificado. Assim que ocorre o evento, o sistema de origem notifica o *endpoint* `HTTP(S)` registrado utilizando o método `POST`, podendo ter um *payload* com as informações da entidade de negócio.

Imagine que um comerciante (*merchant*) abra uma loja virtual dentro da nossa plataforma de e-commerce de exemplo para vender relógios. O comerciante precisa notificar um parceiro externo de serviço de entrega toda vez que um cliente pagou um pedido na sua loja virtual. O comerciante cria um *webhook* na plataforma registrando o *endpoint* `HTTP(S)` do seu parceiro e habilita o evento de *pedido pago*. Toda vez que um cliente desse comerciante pagar um pedido dentro da plataforma, seu parceiro de serviço de entrega será notificado e fará a entrega do pedido. Esse é um cenário comum do uso de *webhook*.

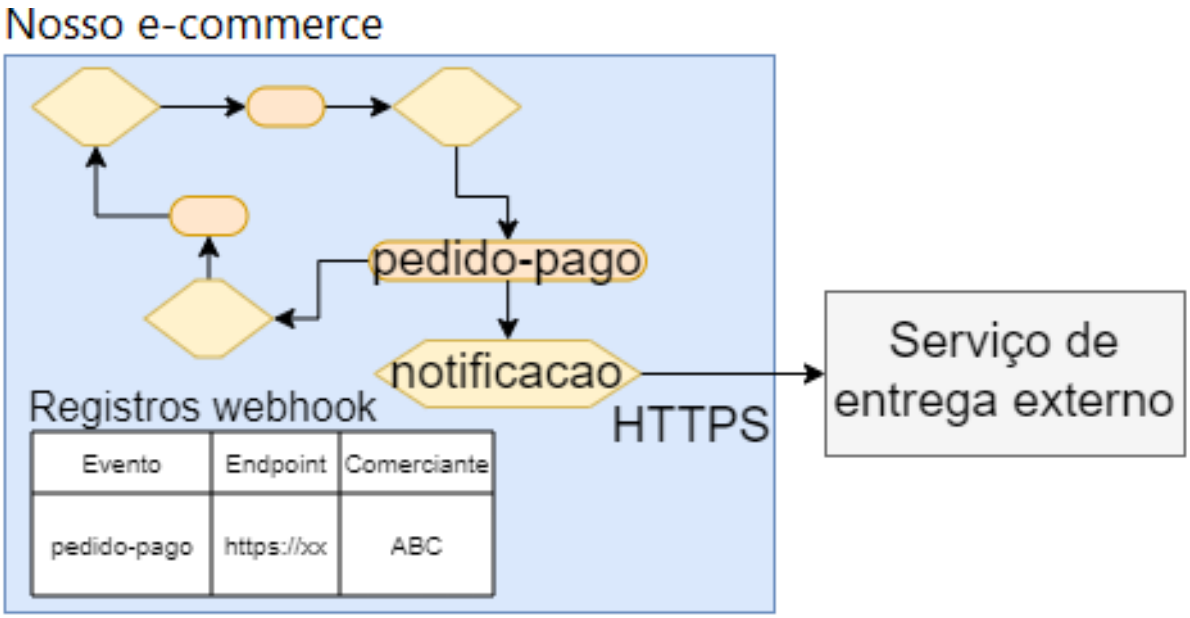

Figura 3.44: Nosso e-commerce notificando um sistema externo com webhook registrado.

Uma grande vantagem de utilizarmos *webhooks* é acionar o *endpoint* registrado apenas quando o evento realmente ocorrer, em vez de o sistema externo fazer *polling* contínuo na plataforma perguntando se ocorreu o evento que lhe interessa. O sistema de origem pode disponibilizar configurações de retentativas.

O exemplo a seguir mostra a diferença de solução utilizando *webhook* e *polling* contínuo entre o sistema de entrega do parceiro do comerciante e nossa plataforma de e-commerce.

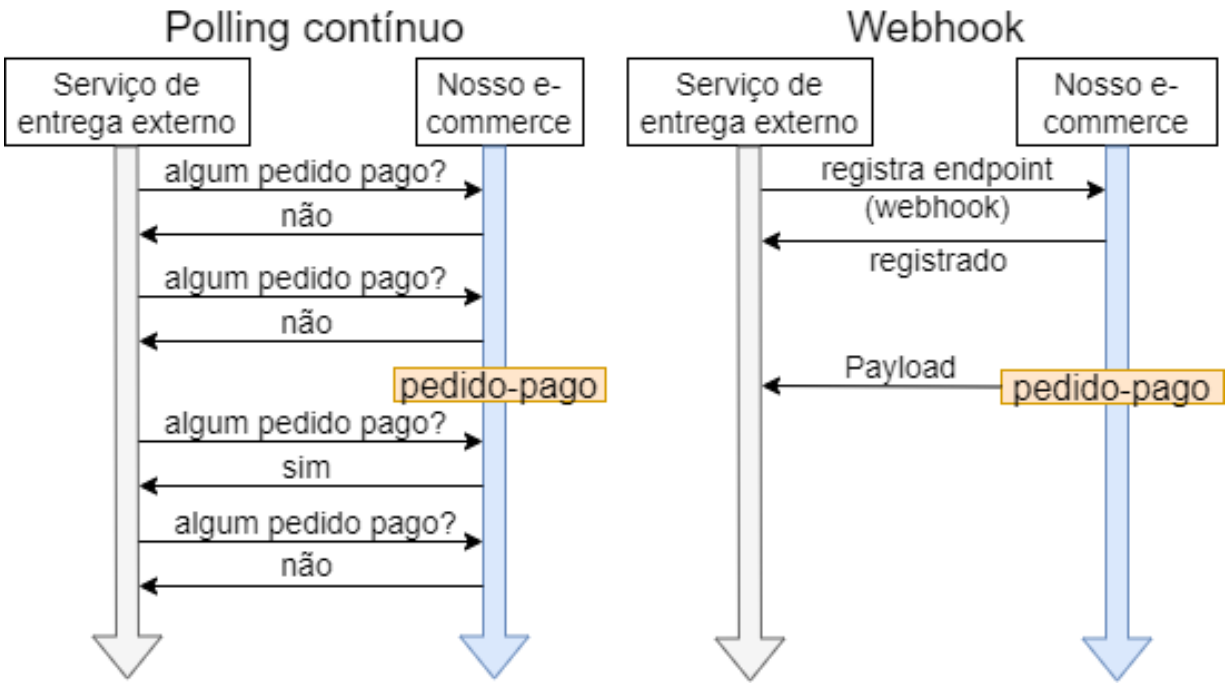

Figura 3.45: Webhook versus polling contínuo.

A seguir, uma lista de algumas plataformas que disponibilizam soluções com *webhooks* ou API de eventos.

- API de eventos do Slack: https://api.slack.com/apis/connections/events-api
- Plataforma do Messenger do Facebook: https://developers.facebook.com/docs/messenger-platform/webhooks
- Shopify: https://shopify.dev/api/admin-rest/latest/resources/webhook
- Stripe: https://stripe.com/docs/webhooks
- GitHub: https://docs.github.com/pt/developers/webhooks-and-events/webhooks
- Discord: https://discord.com/developers/docs/resources/webhook

Os casos de uso que podemos automatizar utilizando *webhooks* são diversos. Veja alguns exemplos:

1. Postar uma nova mensagem em um canal específico do Slack quando um *ticket* for criado no Zendesk com prioridade alta;
2. Criar um pagamento por link no Stripe quando um pedido for criado na plataforma de e-commerce Shopify;
3. Atualizar uma *issue* no Jira quando um *pull request* for criado no GitHub;
4. Postar uma nova mensagem no Discord quando o seu usuário do GitHub for mencionado em um *commit*, comentário ou *pull request*.

3.10 Segurança

Em qualquer estilo arquitetural que adotarmos, independente de ser EDA, precisamos proteger os nossos dados e sistemas de invasores. A segurança da informação é sempre um assunto importante em qualquer solução. Ao migrarmos para o mundo distribuído, adicionamos uma maior complexidade na solução no quesito segurança.

Na arquitetura monolítica (centralizada), basicamente tínhamos poucos pontos de entrada para proteger, por ser uma única aplicação para atingir os objetivos de negócio. Com a arquitetura distribuída, passamos a ter muitos pontos de entrada para proteger, uma vez que temos várias aplicações independentes se comunicando através da rede. Imagine a diferença de complexidade em proteger um prédio com 2 entradas ou um prédio com 20 ou mais entradas.

O objetivo desta seção é dar uma visão geral das funcionalidades relacionadas à segurança com que devemos nos preocupar ao selecionar um *broker* de eventos para nossa solução. Afinal, como veremos nos capítulos seguintes, ele é o coração da EDA.

O *broker* de eventos deve ter a responsabilidade de garantir que um cliente (uma aplicação, dispositivo, pessoa ou integração) esteja autenticado e autorizado para publicar e consumir eventos, criar tópicos, filas, DLQs ou quaisquer outras tarefas administrativas. Deve garantir que os dados trafegados no corpo dos eventos estejam protegidos, tanto os em trânsito quanto os em repouso em seu banco de dados interno. E tudo que acontece no *broker* é rastreável através de registro de auditoria para que possamos obter insights, identificar anomalias e operações não autorizadas, além de atender às políticas de regulamentação da indústria onde a empresa está inserida.

O objetivo e necessidade podem variar de empresa para empresa, mas, no geral, devemos nos preocupar com as seguintes funcionalidades do *broker*:

1. Autenticação;
2. Autorização;
3. Criptografia;
4. Registro de auditoria.

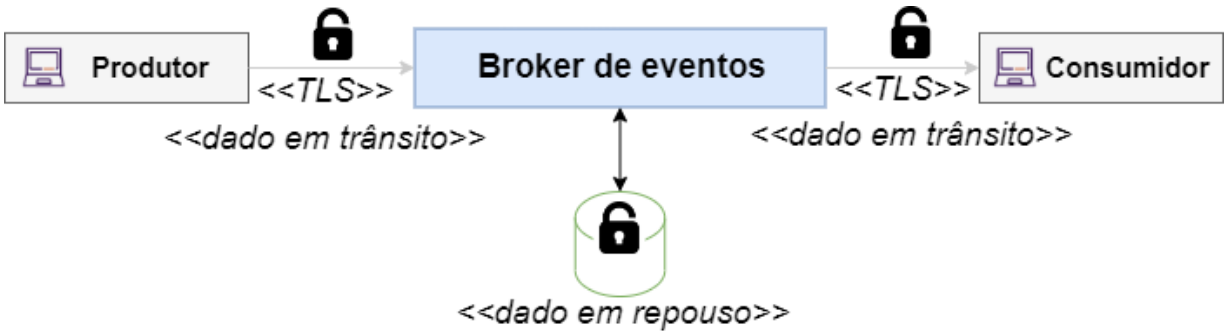

Figura 3.46: Fluxo de publicação e consumo seguro

Autenticação

Autenticação é o processo de confirmação da identidade por meio de algum tipo de credencial. Os clientes precisam provar que eles são quem dizem ser. Precisamos entender que tipo de esquema de autenticação o *broker* suporta e verificar se ele atende à política da

empresa. Tais esquemas podem ser dos mais simples, como *API Key* e *client secret* ou *usuário e senha*, aos mais robustos, com *certificados de cliente* (*Transport Layer Security* – TLS) e integração a diversos provedores de identidade (*identity providers*) — via LDAP, OAuth2 Open ID Connect etc.

Autorização

Autorização é o processo que determina o que um cliente autenticado pode fazer no *broker* de eventos. Nem todos os clientes autenticados podem criar tópicos, por exemplo. Precisamos entender os mecanismos de autorização e o nível de granularidade de controle que o *broker* suporta e verificar se atende à política da empresa.

Um desses mecanismos é o *Access Control List* (ACL), onde podemos controlar o acesso no nível de tópicos e filas, quais tópicos ou filas um determinado cliente pode publicar ou consumir; atribuir clientes a grupos ou diferenciá-los com base em suas funções. Por exemplo, podemos configurar que a aplicação `envio` só pode publicar no tópico `pedido-entregue`. Caso a aplicação `envio` tente publicar no tópico `pedido-criado`, o *broker* não permitirá.

Um outro mecanismo é o OPA (*Open Policy Agent*), que é um motor baseado em política de segurança de propósito geral, leve e *open source* (https://www.openpolicyagent.org/). O OPA permite definirmos nossa política de segurança de forma declarativa, desacoplada e independente de *broker*, plataforma ou ferramentas.

Criptografia

Criptografia é o estudo de como matematicamente codificamos e decodificamos mensagens. Ela impede o acesso não autorizado aos dados (tanto intencional quanto não intencional) e garante a integridade deles. Os eventos podem ser criptografados entre os produtores e o *broker*; dentro do banco de dados interno do *broker*, e entre o *broker* e os consumidores. Precisamos entender se o

broker de eventos tem funcionalidades referentes à criptografia dos dados em trânsito e em repouso.

A forma mais comum de proteger os dados em trânsito é utilizando criptografia de rede, como o *Transport Layer Security* (TLS), também conhecido como **TLS one-way** (apenas o cliente valida se o *broker* é confiável). Além do *TLS one-way*, o *broker* pode suportar o *mutual TLS* (mTLS), também conhecido como **TLS two-way** (tanto o cliente quanto o *broker* validam se ambos são confiáveis). Um ponto importante é que, se habilitarmos o TLS no *broker*, podemos ter algum impacto no desempenho do fluxo de publicação e consumo de eventos, porque é exigido um consumo maior de CPU para criptografar e descriptografar os dados.

Como os *brokers* armazenam os eventos no seu banco de dados internos e têm arquivos de logs, precisamos entender como eles lidam com a criptografia dos dados em repouso. Muitos *brokers* não suportam criptografia em repouso, eles dependem de algum serviço externo para esse fim, como algum serviço de nuvem público ou alguma solução/plataforma para atender a um *broker* instalado no servidor da empresa (*on-premise*).

Como vimos, podemos proteger nossos dados dentro do *broker* com a criptografia em trânsito e em repouso, **mas a melhor forma de proteger os dados é projetarmos o nosso esquema sem dados sensíveis o tanto quanto possível**.

Registro de auditoria

Registro de auditoria contém informações de todas as operações realizadas ou tentativas de operações que ocorrem no *broker* de eventos, seja relacionado a alterações de configurações ou dados de acesso.

Informações de registro de auditoria por si só não são suficientes. Imagine uma aplicação tentando publicar um evento a que ela não tem permissão. Como saberemos que isso está ocorrendo? Esse tipo

de registro precisa ser monitorado para que possamos obter insights, identificar anomalias e operações não autorizadas para tomar as devidas ações o quanto antes. Além disso, precisamos desse tipo de registro para ficar em conformidade com políticas de regulamentações diversas.

Precisamos entender se o *broker* de eventos tem funcionalidades referentes à monitoração, *dashboards*, período de retenção, expurgo e/ou meios de integração com serviços que contemplam essas funcionalidades.

Normalmente, os *brokers* de eventos de mercado atendem aos requisitos de segurança da maioria dos casos de uso. De qualquer forma, não podemos negligenciar o quesito segurança e precisamos nos certificar se o *broker* realmente atende a nossas necessidades. Apesar de o *broker* ser o coração da EDA, ele não é o único componente da solução. Todos os componentes devem estar protegidos contra ameaças e ataques e bem monitorados.

Concluindo

Selecionar o padrão de arquitetura que vamos adotar de acordo com nosso caso de uso é determinante para o sucesso do nosso projeto de EDA. Neste capítulo, entendemos o que é padrão de arquitetura, exploramos os padrões de entrega de evento e padrões de comunicação de mudança de estado que são cruciais dentro de uma solução de EDA. Vimos também padrões relacionados à resiliência com a DLQ e ordenação dos eventos com FIFO. Em seguida, vimos os padrões relacionados à consistência: o SAGA, para manter a consistência do dado através de uma transação que abrange múltiplos serviços, e os padrões *transactional outbox* e CDC, para endereçar o problema da inconsistência ao salvar no banco de dados e publicar um evento.

Vimos também sobre *webhooks*, que são notificações *push* baseadas em `HTTP(S)` acionadas quando um evento ocorre, e sobre as funcionalidades relacionadas à segurança que devemos nos

preocupar quando selecionamos um *broker* para nossa solução, como autenticação, autorização, criptografia e registro de auditoria.

No próximo capítulo, vamos explorar dois assuntos de extrema importância em qualquer projeto de EDA: como podemos identificar os eventos de negócio através de um workshop colaborativo e como podemos documentar nossa solução de EDA utilizando padrões de especificação bem estruturados e amplamente utilizados no mercado.

Capítulo 4
Identificando eventos de negócio e documentando nossa solução

Neste último capítulo da parte de fundamentos, vamos explorar dois assuntos de extrema importância em qualquer projeto de EDA: como identificar eventos de negócio com o *EventStorming* e como documentar nossos ativos de EDA com os padrões de especificações *AsyncAPI* e *CloudEvents*.

4.1 EventStorming

EventStorming (https://www.eventstorming.com/) é um workshop flexível e colaborativo que explora a complexidade do domínio de negócio, com técnicas de modelagem para identificar os eventos e/ou momentos de negócio a partir de um processo de negócio.

Um dos seus principais benefícios – e que, com o tempo, acaba sendo nosso principal objetivo quando pensamos em executá-lo em nossos clientes – é que ele acelera o aprendizado do time sobre o negócio e o processo de negócio em horas, em vez de dias.

No site oficial da ferramenta, o inventor, Alberto Brandolini, destaca que um dos cenários em que usamos o EventStorming é para projetar software orientado a eventos limpo e sustentável, para apoiar negócios em rápida evolução** (EVENTSTORMING, c2023).

Vamos ver um breve resumo de como funciona o EventStorming com alguns dos seus principais artefatos.

No workshop, todo mundo participa: a pessoa desenvolvedora, a especialista do domínio, a arquiteta de solução, pessoas com papéis que podem contribuir e a facilitadora. A pessoa facilitadora tem o

objetivo de manter o grupo focado e engajado, guiando-os na realização da modelagem completa do domínio. Não existe uma regra de quantidade de pessoas e tempo de duração.

O grupo inicia com a identificação dos **eventos de domínio** (*domain events*), que são nossos eventos ou momentos de negócio, escrevendo-os em um *post-it* e colocando-os no quadro na sequência da linha do tempo. Esse processo é contínuo. É papel da pessoa facilitadora fazer o *merging* dos diferentes pontos de vista, tentar capturar qualquer sinal de atenção, conflito, comentários sobre um determinado ponto do processo de negócio, escrever em um *post-it* e colocar ao lado do *post-it* do evento relacionado como um ***hot spot***.

O grupo então adiciona **comandos** (*commands*) ou *triggers* no quadro, que são ações que ativam eventos de domínio. O comando pode ser ativado pelo usuário através de um botão, por um sistema externo ou até mesmo por outros eventos de domínio.

Depois, o grupo identifica os **agregados** (*aggregates*) e os agrupam em um **contexto delimitado** (*bounded contexts*). Um agregado é um conjunto de entidades (*entities*) e objetos de valor (*value objects*) que são tratados como uma unidade, definindo limites da consistência dentro do contexto delimitado.

Finalmente, os relacionamentos entre os contextos delimitados são adicionados para criar um **mapa de contexto** (*context map*). O resultado do modelo é o ponto de partida para pensarmos em desenvolvimento de código, baseado em uma linguagem comum entre os times dentro de um contexto delimitado — a **linguagem ubíqua** (*ubiquitous language*).

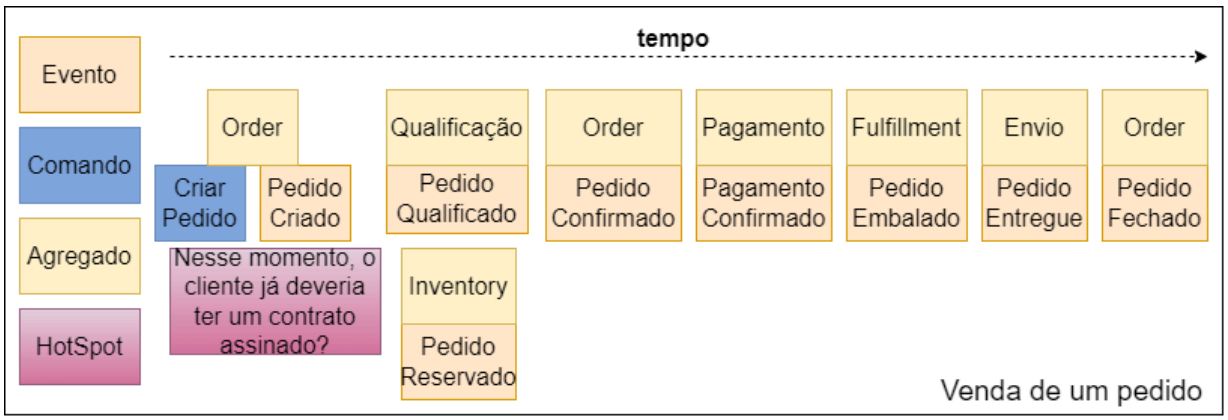

Figura 4.1: Exemplo simples de um quadro do EventStorming.

Lições aprendidas na execução do EventStorming na empresa ABC

Foi necessário rodarmos 10 sessões do workshop até que todos estivessem confortáveis com o entendimento do domínio. Cada sessão durou cerca de 1h por dia. Foi um aprendizado enriquecedor para todos da empresa ABC. Como resultado das sessões de EventStorming, chegamos em quase 30 eventos de domínio e muitos *hotspots*, oportunidades e agregados. Veja algumas das lições aprendidas nessas 10 sessões do nosso workshop:

1. A empresa não tinha um processo de negócio claro e bem documentado. Isso fez com que as sessões do workshop se estendessem;
2. O processo de negócio parecia claro para todos, até colocarmos diferentes especialistas de domínio na mesma sala para discutir o processo;
3. Não vale a pena realizar o ciclo de EventStorming sem as pessoas-chaves, que realmente entendem do domínio e tomam decisões;
4. Conseguir agenda das pessoas-chaves para executar as sessões do workshop é muito difícil. A lição aprendida aqui é que, para projetos prioritários e estratégicos, esse problema é mitigado;
5. A cada sessão, era criado um desenho com o que foi levantado e os pontos importantes (*hotspots*). Não queríamos correr o

risco de alguém apagar o quadro branco e confiar apenas nas fotos tiradas a cada sessão;
6. Quando ocorriam discussões desnecessárias, é importante o papel do facilitador, para fazer as pessoas voltarem para o que realmente interessa.

No meu entendimento, o EventStorming é o pontapé inicial do levantamento e refinamento da solução de EDA. Com o domínio do negócio mais bem compreendido, a solução vai sendo aprimorada.

Dificilmente teremos uma solução completa após o EventStorming no início do projeto, mas ele acelera muito o entendimento e o ritmo no decorrer do projeto — e nada impede também de fazer novas sessões, caso tenha alguma parte do domínio ainda mal entendida.

Com os eventos de negócio ou momentos de negócio identificados (pelo menos parte deles), precisamos definir seu esquema e documentá-los.

4.2 AsyncAPI

AsyncAPI (https://www.asyncapi.com/) é uma especificação para definir interfaces orientadas a eventos de maneira estruturada dentro da nossa solução de EDA. Ela é independente da linguagem de programação das aplicações produtoras e consumidoras, do *broker* de eventos e protocolos. AsyncAPI é para a EDA o que a OpenAPI é para REST API.

Criamos um ou mais arquivos `JSON` ou `YAML` de acordo com as regras da especificação. Eles podem ser usados para gerarmos a documentação e o código de acordo com a nossa linguagem de programação. Além disso, podemos usar como nosso catálogo de eventos.

Vamos criar a nossa documentação AsyncAPI de acordo com nossa solução EDA de um e-commerce da empresa fictícia ABC.

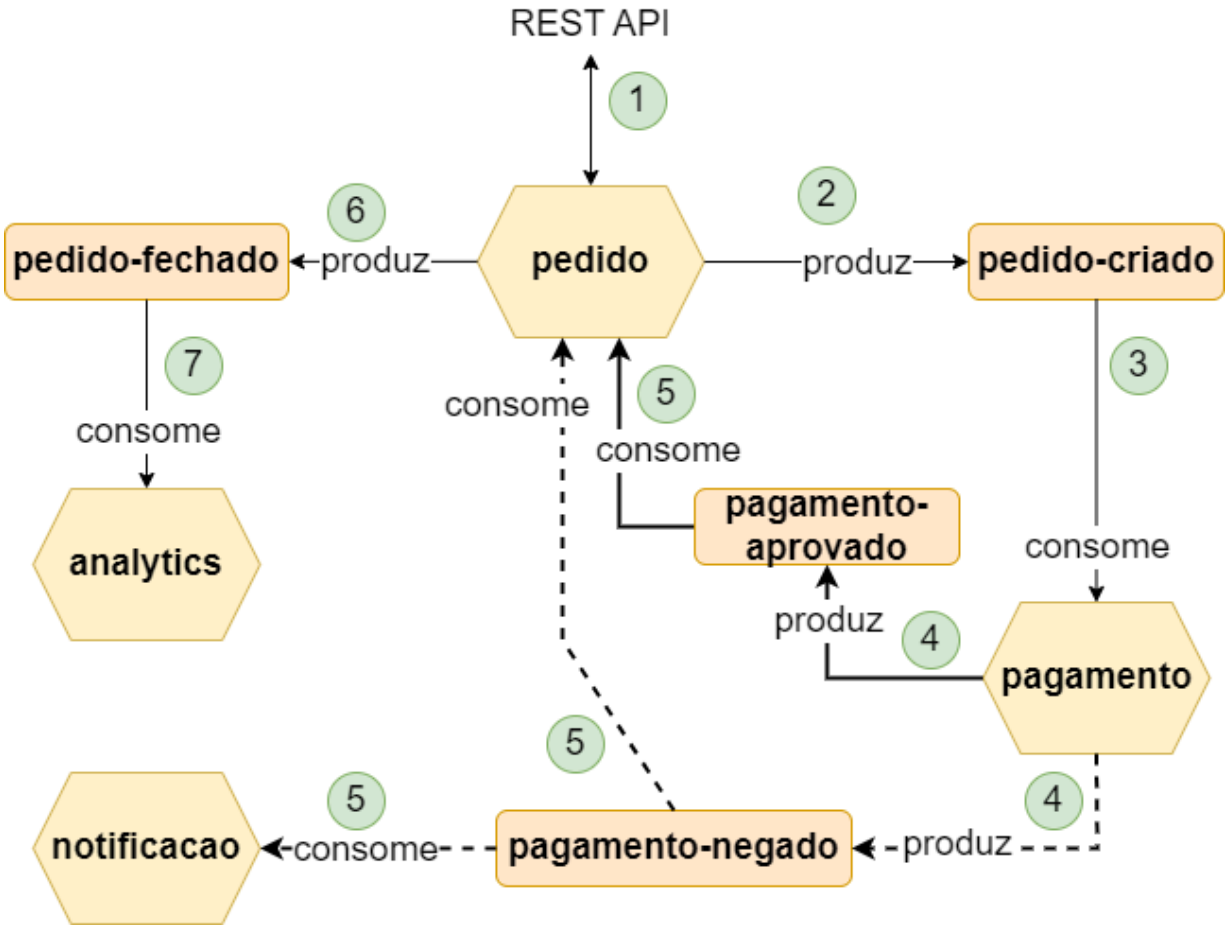

Figura 4.2: Exemplo fictício de uma solução e-commerce da empresa ABC.

1. O usuário realiza um pedido;
2. A aplicação `pedido` faz as devidas validações e publica o evento `pedido-criado`;
3. A aplicação `pagamento` consome o evento `pedido-criado` para processar o pagamento do pedido. Ele é o responsável por realizar o pagamento no *gateway* de pagamento (omitido no nosso desenho);
4. A aplicação `pagamento` informa se o pagamento foi aprovado, através do evento `pagamento-aprovado`, ou reprovado, através do evento `pagamento-negado`;

5. Após o pagamento ser aprovado ou negado, a aplicação `pedido` atualiza a situação do pedido de acordo; se o pagamento for negado, a aplicação `notificacao` notifica o cliente de que algo de errado ocorreu com seu pedido;
6. A aplicação `pedido` conclui o pedido e publica o evento `pedido-fechado`;
7. A aplicação `analytics` consome o evento `pedido-fechado` para gerar análises e relatórios diversos.

Está longe de ser uma solução realista de e-commerce, mas já conseguimos passar a ideia da documentação da nossa solução de EDA através da especificação AsyncAPI. Nesse exemplo fictício, o fluxo é iniciado externamente através de uma chamada síncrona REST API. Temos quatro aplicações e quatro eventos envolvidos na nossa solução EDA para processar os pedidos dos clientes da empresa ABC. As setas indicam a direção do fluxo, se a aplicação está produzindo ou consumindo o evento.

As setas pretas mais largas correspondem ao caminho feliz, com o pagamento aprovado. Já as setas pontilhadas correspondem ao fluxo onde o pagamento foi negado. As outras setas correspondem ao fluxo comum.

Utilizei o **AsyncAPI Studio** (https://studio.asyncapi.com/) para criar o arquivo e, ao mesmo tempo, visualizar o resultado:

```
asyncapi: '2.6.0'
info:
  title: Ecommerce
  version: '1.0.0'
channels:
  pedido-criado:
    publish:
      tags:
        - name: pedido
      description: "Evento gerado quando um pedido for criado."
      message:
        name: order
```

```
        $ref: '#/components/messages/order'
    subscribe:
      tags:
        - name: pagamento
      description: "Consome esse evento quando for necessário processar um
pedido recém criado."
      message:
        name: order
        $ref: '#/components/messages/order'
  pagamento-aprovado:
    publish:
```

O arquivo `.yaml` completo está disponível no nosso projeto do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap4/asyncapi). O *screenshot* a seguir é um trecho da documentação gerada. Pegue o arquivo completo no GitHub e cole-o no AsyncAPI Studio para ver o resultado completo da documentação.

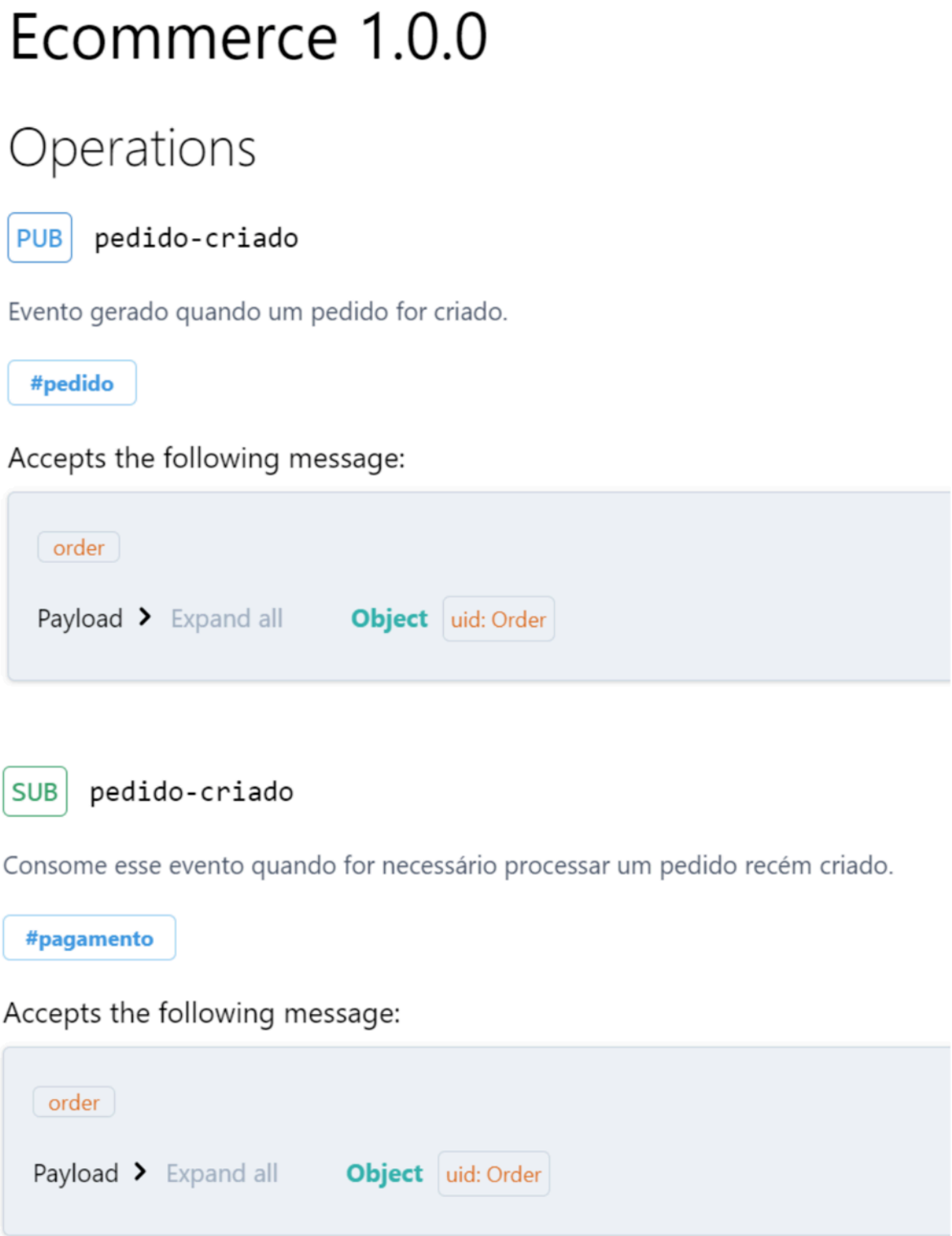

Figura 4.3: Parte da especificação AsyncAPI da nossa solução fictícia de e-commerce.

Nesse arquivo que criamos, definimos os campos: `asyncapi`, que tem informação da versão da especificação; `info`, que tem como campos obrigatórios o título da solução e a versão da solução; `channels`, que corresponde aos eventos da nossa solução, bem como suas descrições e informações das aplicações que produzem e consomem desses eventos; e `components`, que tem a definição da estrutura do corpo dos eventos e que é referenciado em cada evento definido no campo `channels`.

Duas estratégias comuns na organização dos arquivos são: criar um arquivo por aplicação ou por domínio do negócio. Por aplicação, cada arquivo terá informação dos eventos que ela publica e subscreve. No nosso exemplo, teremos um arquivo para a aplicação, por exemplo, `analytics` contendo informações do evento que ele consome, `pedido-fechado`. Já por domínio, o arquivo é criado de acordo com um conjunto de aplicações dentro do seu contexto de negócio. Por exemplo, o e-commerce da empresa fictícia ABC é um domínio e as quatro aplicações são os subdomínios.

Na maioria dos projetos de que participo, crio um arquivo por domínio para expor em um portal do desenvolvedor (*developer portal*). A empresa pode ter vários outros domínios que precisam consumir eventos uns dos outros. Além disso, se criarmos um único arquivo com todos os domínios, ele poderá ficar confuso e difícil de manter.

4.3 CloudEvents

CloudEvents (https://cloudevents.io/) é, conforme está em seu website, uma especificação para **descrever dados de eventos em formatos comuns** para fornecer interoperabilidade entre serviços, plataformas e sistemas. Ele é parte da Cloud Native Computing Foundation, a CNCF (https://www.cncf.io/). Ele nos permite uma

definição comum dos dados no corpo da mensagem, que facilita a comunicação entre aplicações em diferentes plataformas.

De acordo com a especificação, temos os campos obrigatórios: `id`, que serve como identificador do evento; `source`, que identifica o contexto do qual o evento ocorreu; `specversion`, que é a versão usada da especificação do CloudEvents; e, por fim, `type`, que descreve o tipo do evento. Existem outros campos opcionais, mas vale destacar o campo `data` (https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md), que corresponde ao conteúdo do corpo da mensagem.

Vamos atualizar nosso arquivo AsyncAPI criado anteriormente para adaptá-lo à especificação CloudEvents. Repare que utilizamos uma referência externa da especificação. Poderíamos adicionar os campos manualmente direto no arquivo:

```
schemas:
    OrderSchema:
      type: object
      schemaFormat: 'application/cloudevents+json; version=0.2;
charset=utf-8'
      allOf: [{
        $ref:
'https://raw.githubusercontent.com/cloudevents/spec/v1.0.1/spec.json'
      }, {
        $ref: '#/components/schemas/Data'
      }]
```

Usamos o formato `.json` no corpo da mensagem em nosso exemplo, mas a especificação nos permite utilizar outros formatos, como Apache Avro e ProtoBuf. O arquivo `.yaml` atualizado com os campos definidos pelo CloudEvents está disponível no nosso projeto do GitHub.

A especificação AsyncAPI e o CloudEvents podem ser usados em conjunto, uma não substitui o outro. Utilizar padrões bem estruturados e amplamente utilizados no mercado nos ajuda a

ganhar velocidade no desenvolvimento e a comunicar melhor nossa solução.

Concluindo

Neste capítulo, vimos como o EventStorming pode nos ajudar na identificação dos eventos de negócios, entre outros artefatos. Em seguida, vimos como podemos documentar nossos ativos de EDA utilizando AsyncAPI e CloudEvents. No próximo capítulo, começa a parte II do livro, onde vamos explorar as principais características e funcionalidades de um *broker* de eventos, que é considerado o coração da EDA.

Parte II: Broker de eventos

CAPÍTULO 5
Broker de eventos habilita a Arquitetura orientada a eventos

Começamos a parte II do livro! Aqui, vamos explorar as principais características e funcionalidades de um *broker* de eventos, que é considerado o coração da EDA.

Neste capítulo, vamos explorar o papel do *broker* de eventos na EDA, além de alguns assuntos importantes relacionados a ele, como os padrões de implantação (*deploy*), a operação e governança e, por fim, o gerenciamento de esquemas.

5.1 O papel do broker de eventos na Arquitetura orientada a eventos

Como vimos na seção *Componentes-chaves* do capítulo 2, *Entendendo os principais conceitos*, o *broker* de eventos é uma solução/software que atua como uma ponte entre o produtor do evento e o consumidor do evento.

O *broker* de eventos tem um papel fundamental na EDA: é ele que garante o desacoplamento entre as aplicações com seu suporte ao modelo *publish-subscribe*. Por isso, ele é considerado o coração da EDA.

As principais responsabilidades de um *broker* de eventos são:

- Gerenciar conexões das aplicações produtoras e consumidoras;
- Gerenciar o controle de acesso dos usuários e aplicações;

- Gerenciar esquemas, filas, aplicações e tópicos, que são as estruturas lógicas de agrupamento dos eventos;
- Gerenciar a persistência dos eventos publicados;
- Garantir a entrega dos eventos para as aplicações consumidoras;
- Garantir alta disponibilidade e que seja tolerante a falhas;
- Garantir o desempenho do fluxo de publicação e consumo de eventos de acordo com a demanda;
- Gerenciar, monitorar e observar o fluxo dos eventos.

Uma empresa que adota a EDA busca basicamente **capturar e processar os eventos de negócio ou momentos de negócio em tempo real de maneira desacoplada e adaptativa**. É através do *broker* de eventos que a EDA brilha.

Se pudéssemos resumir em duas as principais responsabilidades de um *broker* de eventos, elas seriam:

1. O *broker* deve garantir que esteja disponível para que as aplicações produtoras publiquem os eventos de negócio ou momentos de negócios capturados;
2. Uma vez que o *broker* confirmou o recebimento do evento, ele deve garantir que o evento seja entregue para as aplicações consumidoras registradas, mesmo em cenário de falha.

5.2 Padrões de implantação (deploy)

Como qualquer software, o *broker* de eventos pode ser implantado (*deploy*) na infraestrutura do cliente (*on-premise*) ou na infraestrutura de algum provedor de nuvem, como a AWS, Microsoft Azure ou Google Cloud.

O tipo de implantação que um fornecedor disponibiliza para seu *broker* determina a maneira como os clientes vão configurá-lo e gerenciá-lo. Os padrões de implantação que veremos a seguir

podem ser divididos basicamente em dois tipos: **nuvem (*cloud*)** ou **software**.

> A escolha do padrão de implantação pode variar de acordo com custo e/ou requisitos de conectividade de rede e/ou capacidade da empresa de autogerenciar um *broker* juntamente aos requisitos não funcionais da solução, tais como segurança, conformidade com alguma regulamentação da indústria na qual a empresa está inserida, escalabilidade, elasticidade, *throughput*, latência, resiliência, entre outros.

Nuvem (Cloud)

O *broker* pode ser implantado/disponibilizado em um ambiente de nuvem compartilhada (*multi-tenant*) ou dedicada (*single-tenant*). O ponto-chave aqui é que o cliente utiliza seu *broker* e armazena seus dados na nuvem e a infraestrutura é gerenciada por esses provedores.

Vamos explorar dois padrões de implantação do *broker* na infraestrutura da nuvem: o *broker serverless* e o *broker gerenciado*.

Broker Serverless

São *brokers/clusters* provisionados e gerenciados pelo provedor de nuvem e precificados baseado no uso (o cliente paga somente o que usa). Não há contrato de longo prazo, taxas ou compromissos mínimos. Esse tipo de *broker* tem seus recursos compartilhados entre clientes (*multi-tenant*) do provedor da nuvem, seguindo o modelo *software como serviço*.

Esse padrão é ideal nos casos em que o cliente está experimentando uma ideia, não conhece sua carga de trabalho, tem tráfego de eventos/dados imprevisíveis ou não consegue gerenciar a capacidade de uso. Normalmente, esse padrão de implantação é um

problema no quesito segurança e regulamentações diversas, devido ao compartilhamento de recursos.

Broker gerenciado

São *brokers/clusters* gerenciados pelo provedor de nuvem, e o cliente provisiona os recursos computacionais que executam o *broker*. Além de provisionar, os fornecedores dos *brokers* deixam o cliente ter autonomia em algumas configurações de escalabilidade, disponibilidade e recuperação de desastre do *broker/cluster*. A precificação é baseada nos recursos computacionais utilizados e no tempo de execução desses recursos. Esse tipo de *broker* não tem seus recursos compartilhados (*single-tenant*) entre clientes do provedor da nuvem.

Esse padrão é ideal nos casos em que o cliente precisa ter mais controle e visibilidade dos recursos computacionais utilizados pelo *broker*. O cliente precisa conhecer sua demanda de carga de trabalho, por exemplo, se seu tráfego de eventos é conhecido e consistente. Esse padrão pode ajudar na gestão financeira do cliente, possibilitando uma melhor previsibilidade do custo.

Se usado massivamente, o *broker gerenciado* tende a ser mais barato que o *broker serverless*, mas talvez o cliente precise de um investimento inicial maior e de um contrato de médio a longo prazo.

Software

O *broker* pode ser implantado/disponibilizado na infraestrutura do cliente (*on-premise*) ou na infraestrutura de algum provedor da nuvem. O ponto-chave aqui é que o cliente é o responsável por implantar e gerenciar a infraestrutura do seu *broker/cluster*.

Vamos explorar dois padrões de implantação do *broker* como *software*: o *broker em máquina virtual* e o *broker conteinerizado*.

Broker em máquina virtual

O *broker* é instalado em máquina virtual e pode ser utilizado *on-premise* ou na nuvem. Por exemplo, o cliente pode implantar seu *broker/cluster* em uma máquina virtual da AWS, Amazon Elastic Compute Cloud (EC2). Mesmo na nuvem, o cliente é o responsável pela implantação, manutenção, atualização etc.

Broker conteinerizado

O *broker* é instalado em contêiner e gerenciado por um orquestrador de contêineres, como o Kubernetes. Com um orquestrador de contêineres, conseguimos automatizar a implantação e a escalabilidade do *broker*, além de ser mais portátil, com menos dependência do fornecedor. O cliente pode implantar seu *broker* na nuvem ou *on-premise*. O cliente pode também utilizar qualquer plataforma compatível com Kubernetes, como Red Hat OpenShift, Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Google Kubernetes Engine (GKE), entre outras.

Multi-cloud e híbrida

Até então, temos tratado a implantação do *broker* de eventos na nuvem ou *on-premise*. Há casos em que precisamos que nossa solução de EDA (*broker/cluster*) seja mais distribuída em múltiplas redes, *datacenters*, localizações ou nuvens para atingir os objetivos de negócios. Nesse cenário, temos uma junção dos dois tipos que vimos anteriormente, *nuvem* e *software*. Nossa solução de EDA passa a ser de *múltiplas nuvens* (*multi-cloud*) ou *híbrida* (*hybrid cloud*).

Multi-cloud: A solução de EDA (os *brokers* da solução) está em múltiplas nuvens públicas, como AWS, Microsoft Azure e Google Cloud.

Híbrida: A solução de EDA (os *brokers* da solução), além de estar em uma ou mais nuvens públicas, também está *on-premise*.

Imagine que na nossa empresa temos aplicações produtoras e consumidoras na nuvem da AWS e Microsoft Azure, além de termos aplicações *on-premise*. Um evento publicado por uma aplicação hospedada na infraestrutura da AWS pode ser o gatilho para aplicações hospedadas na infraestrutura da Microsoft Azure e *on-premise*. Esse é o cenário de uma solução híbrida. Não importa onde suas aplicações estejam, o cliente vai conseguir distribuir seus dados através dos *brokers* de eventos entre várias nuvens e *on-premise*.

Uma solução híbrida nos dá uma maior agilidade e eficiência no compartilhamento dos dados entre nossas aplicações *on-premise* e nativas da nuvem, mas ela traz um grande desafio na distribuição desses eventos. Precisamos de uma maneira com que nossa solução híbrida de EDA seja tratada como uma única rede. Aí entra o conceito de *rede de eventos* (do inglês, *event mesh*).

Com *event mesh*, podemos configurar múltiplos *brokers/clusters* para fazerem parte da mesma rede de eventos e, com isso, possibilitar que eventos sejam publicados e consumidos de qualquer infraestrutura. *Event mesh* possibilita outras funcionalidades além de habilitar essa comunicação entre *brokers/clusters*, como dinamicamente fazer o roteamento dos eventos e aplicar políticas de segurança em nível de rede.

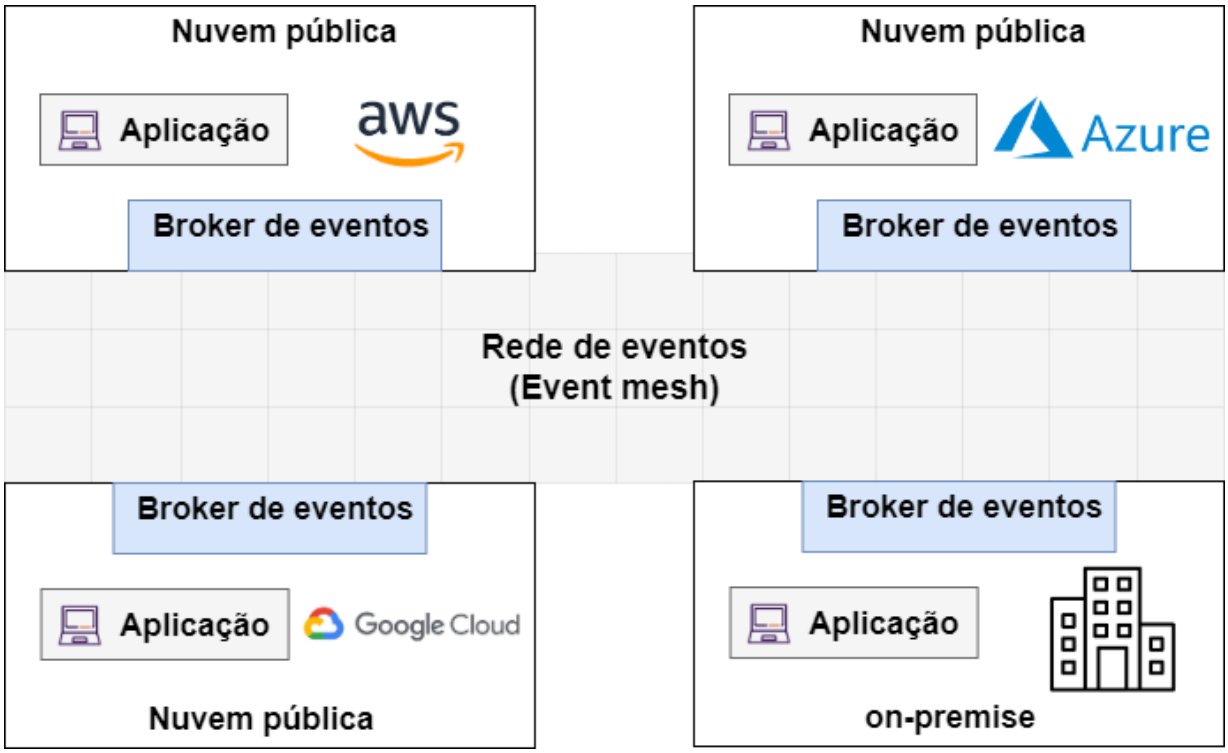

Figura 5.1: Distribuição de eventos como uma única rede de eventos (event mesh).

Nem todos os fornecedores de *broker* têm uma solução de *event mesh*. A forma mais comum é a replicação dos eventos entre *brokers/clusters* para garantir uma alta disponibilidade dos dados e evitar a perda dos dados caso o *broker* caia. Uma opção para fazer a replicação dos eventos é utilizar conectores, como o Kafka Connect ou similares.

Existem vários aspectos que influenciam a escolha do padrão de implantação. Essa escolha vai variar de empresa para empresa de acordo com os requisitos que mencionamos anteriormente.

5.3 Operação e governança

As capacidades do *broker* para operá-lo referem-se às tarefas do dia a dia de gerenciamento. Vamos explorar algumas das

funcionalidades que podem ser oferecidas pelo fornecedor do *broker* para operá-lo: *interfaces de interação com o broker*, *monitoração*, *métricas de negócio* e *criação e visualização da execução da coreografia de eventos*.

Interfaces de interação com o broker

Basicamente temos três opções de interação com o *broker*: *Console*; *interface de linha de comando* (do inglês, *command-line interface*, ou simplesmente *CLI*) e *APIs administrativas*. Vamos explorar uma quarta opção, a biblioteca de cliente/SDK, no próximo capítulo.

É importante analisar quais dessas opções o fornecedor do *broker* disponibiliza. Essas opções permitem que criemos e configuremos nossos ativos (eventos/tópicos, aplicações, esquemas etc.) manualmente e/ou utilizando práticas de DevOps, como automações e infraestrutura como código (IaC).

Por exemplo, podemos criar um tópico, *pedido-criado*, manualmente, pelo console da Confluent:

New topic

General

Topic name* ⓘ
pedido-criado

Partitions* ⓘ
6

Storage

Cleanup policy ⓘ
Delete

Retention time ⓘ
1 week

Retention size ⓘ
Infinite

Message size

Maximum message size in bytes ⓘ
2097164 bytes

Other Settings

cluster	cluster_0
replication.factor	3
min.insync.replicas	2
compression.type	producer
leader.replication.throttled.replicas	
message.downconversion.enable	true

Cancel

Figura 5.2: Screenshot da tela de criação de tópico no Confluent.

Podemos automatizar a criação do tópico *pedido-criado* através do CLI da Confluent:

```
confluent kafka topic create pedido-criado --partitions 6
```

Ou podemos automatizar através da API administrativa da Confluent:

```
curl --data "{"topic_name": "pedido-criado","partitions_count": 6}" -H "Content-Type: application/json" -H "Authorization: Basic <<cluster_api-key>>" --ssl-no-revoke -X POST https://pkc-00000.region.provider.confluent.cloud:443/kafka/v3/clusters/ktd-ymyk09/topics
```

Monitoração

Podemos definir **monitoração** como o processo de capturar o comportamento de um sistema ao longo do tempo. Isso pode ser feito através de *verificações de integridade* (*health checks*) e de *métricas*.

As verificações de integridade verificam a saúde do sistema e dos seus componentes periodicamente. Em alguns casos, o próprio sistema ou seus componentes conseguem se recuperar automaticamente (*self-healing*). As métricas coletadas, sejam elas específicas do *broker*, dos recursos computacionais onde o *broker* é executado, das aplicações, das autorizações etc., nos ajudam a monitorar o desempenho e o uso do sistema, bem como a identificar anomalias diversas.

Como qualquer software, o *broker/cluster* precisa ser monitorado. Temos que avaliar se o fornecedor do *broker* oferece uma robusta solução de monitoração para todas as aplicações conectadas a ele, tópicos ou quaisquer outros ativos que planejamos usar. Uma monitoração contínua é essencial para identificarmos e respondermos a potenciais falhas na nossa solução.

Sistemas de monitoramento normalmente têm a capacidade de emitir alertas de acordo com as métricas oferecidas. Por exemplo, o time de operações pode receber um alerta para executar alguma ação caso o tópico *pedido-criado* tenha mais de 500 eventos acumulados ou não consumidos, por exemplo, ou se alguma aplicação consumidora perdeu a conectividade com o *broker*.

Os fornecedores de *brokers* podem ter sua própria solução de monitoração. Eles podem oferecer monitorações básicas ou avançadas, dependendo do tipo de plano contratado, bem como possibilidades de integração com outras soluções de monitoração de mercado, *open source* ou não, tais como:

- AppDynamics: https://www.appdynamics.com/
- DataDog: https://www.datadoghq.com/
- Dynatrace: https://www.dynatrace.com/
- Prometheus: https://prometheus.io/

Se sua empresa já utiliza uma dessas soluções de monitoração de mercado, talvez seja interessante integrá-la à sua solução de EDA, incluindo o *broker*, para ter uma monitoração centralizada.

Métricas de negócio

Vimos anteriormente a importância das métricas coletadas para identificar anomalias diversas. Mas existe um outro tipo de métrica tão importante quanto as outras: a métrica de negócio. Temos que avaliar se o fornecedor do *broker* nos oferece alguma forma de coletar esse tipo de métrica. Essas métricas nos darão *insights* para aprimorar o negócio.

As métricas de negócio nos ajudam a direcionar melhor nossos esforços para atender à necessidade dos nossos clientes. Através dessas métricas, podemos monitorar e analisar o comportamento dos nossos clientes e do nosso negócio. Em uma solução de *e-commerce*, por exemplo, responder a perguntas do tipo: *quantos pedidos/eventos foram concluídos em um determinado período?*;

qual a média de tempo que um pedido leva para ser entregue/concluído?; *onde está sendo o gargalo no fluxo?*; ou, ainda, *quais produtos são mais vendidos em um determinado período ou região?* As possibilidades são diversas.

Criação e visualização da execução da coreografia de eventos

Muitos projetos de EDA são baseados na coreografia de eventos (cadeia de eventos). Criar e visualizar a coreografia no *broker* e termos a *The Big Picture* é difícil. Fornecedores de *broker* que oferecem esse tipo de capacidade simplificam a operação, por permitir que criemos nosso fluxo de publicação e consumo em uma visão única.

A visualização da coreografia pode ser disponibilizada em tempo de execução para nos ajudar na rastreabilidade da execução em tempo real, aumentando nossa capacidade de resolução de problemas. Imagine conseguir identificar um problema de uma transação específica desde o *pedido-criado* até o *pedido-fechado* de uma coreografia complexa com centenas de eventos e aplicações de forma intuitiva.

O *broker* também pode oferecer meios de criarmos nossa coreografia pelo *CLI*. Podemos importar um arquivo AsyncAPI e o *broker* criará nossa coreografia e catálogo automaticamente.

A seguir, um *screenshot* do *Designer* do portal de eventos da Solace. Conseguimos criar e configurar nossa coreografia de eventos através de uma tela gráfica:

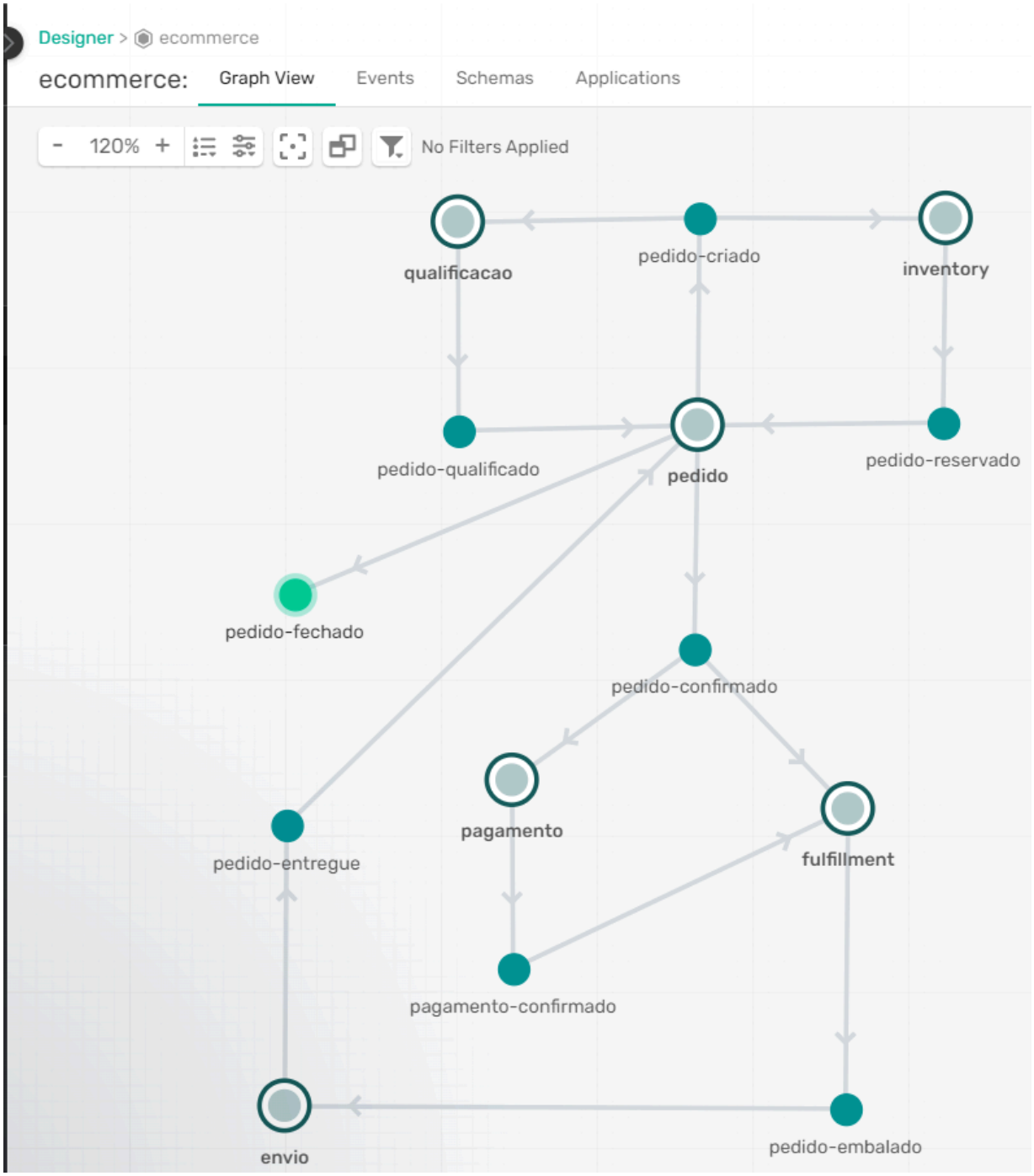

Figura 5.3: Screenshot da Designer do portal de eventos da Solace.

Essa capacidade é atualmente considerada emergente e poucos fornecedores oferecem. Além da Solace, a Confluent (*Stream Designer*) também oferece esse tipo de capacidade.

Avaliar as capacidades do *broker* para operá-lo antes de adquiri-lo é muito importante. Por exemplo, um *broker* que não oferece outras opções de interação além do console impossibilita aplicar automação ou qualquer outro princípio de DevOps. Ter um *broker* com monitorações avançadas, com as quais é possível obter insights de negócios através das métricas geradas por ele, além de possibilitar a resolução de problemas de forma ágil através de uma visualização única da execução do fluxo de eventos em tempo real, é um grande diferencial e pode ser requisito essencial para nosso negócio.

Governança

À medida que nosso negócio cresce, nossa solução de EDA passa a ter mais e mais ativos (eventos/tópicos, aplicações, esquemas etc.). Um dos principais desafios que enfrentamos é como governar nossos ativos dentro de um contexto de constante crescimento. As capacidades do *broker* para governá-lo referem-se às funcionalidades para descobrir, compreender, organizar e reutilizar os ativos de maneira fácil e ágil. Vamos explorar algumas das funcionalidades que podem ser oferecidas pelo fornecedor do *broker* para governá-lo: *catálogo* e *portal do desenvolvedor*.

Catálogo

Um **catálogo** nada mais é do que um repositório centralizado para organizar nossos ativos de EDA. Toda vez que criamos um ativo (por exemplo, um evento/tópico), ele automaticamente é disponibilizado no catálogo com todas suas informações e relacionamentos.

O fornecedor de *broker* oferece formas de realizarmos pesquisas avançadas no catálogo para compreender e aumentar a possibilidade de reúso dos nossos ativos pelos times. Ele permite que tenhamos a resposta para perguntas do tipo "*O que significa o evento de negócio* `pedido-fechado` *e quais aplicações dentro do meu ecossistema de e-commerce o publicam e o consomem?*".

Portal do desenvolvedor

No nosso contexto de EDA, o fornecedor do *broker* de eventos pode oferecer o **portal do desenvolvedor** de eventos. O portal é um canal de autoatendimento (*self-service*) para os desenvolvedores. Ele permite que desenvolvedores tenham acesso a informações e documentação necessárias (conectar ao *broker*, controle de acesso, formato da mensagem, protocolos de mensagens disponíveis etc.) para que construam as aplicações que vão publicar e consumir os eventos de negócio. Além disso, ele permite que pessoas desenvolvedoras gerem códigos nas principais linguagens de programação a partir da definição do AsyncAPI, aumentando a produtividade do time.

Quando estamos iniciando um projeto de EDA, normalmente não nos preocupamos com a governança. Mas quando o projeto vai crescendo, governá-lo vai ficando cada vez mais desafiador. Para piorar, vários outros times dentro da organização podem ter seus projetos de EDA. Ter uma visão única de todos os ativos de todos os projetos de EDA facilita a descoberta, compreensão, organização e reutilização desses ativos entre projetos. Minha recomendação é avaliar as capacidades do *broker* para governá-lo desde o primeiro projeto de EDA na organização.

5.4 Gerenciamento de esquema

Como vimos na seção *Formato da mensagem* do capítulo 2, *Entendendo os principais conceitos*, uma mensagem ou evento tem um formato e pode ter um esquema definido para que as aplicações consigam interpretar e validar a estrutura do corpo do evento de acordo com as regras definidas do esquema.

Alguns formatos não precisam de um esquema por serem autodescritivos, como `JSON` e `XML` — apesar de terem seus respectivos esquemas, *JSON esquema* e o *XML esquema* (`XSD`), respectivamente. Outros formatos obrigatoriamente precisam ter um

esquema definido para serem serializados e desserializados, como os formatos binários Apache Avro e ProtoBuf, que vimos na seção *Formato da mensagem* do capítulo 2.

O fato é que podemos ter um fluxo de publicação e consumo de eventos sem nenhum tipo de contrato ou validação da estrutura do corpo dos eventos. Isso cria um contrato implícito entre as aplicações produtoras e consumidoras. Há uma suposição do produtor de que o consumidor entenderá o formato da mensagem, enquanto o consumidor também assume que o produtor continuará enviando as mensagens no mesmo formato. Mas a realidade é que as estruturas sofrem evoluções ou versionamentos com o tempo para acompanhar a necessidade do negócio e temos que garantir que as aplicações estejam preparadas para essa evolução e não deixem de funcionar.

Esquemas são como um contrato, que tem um conjunto de termos que garante que as aplicações consigam interpretar e validar os dados do corpo dos eventos. Isso faz com que o fluxo de publicação e consumo de eventos fique mais resiliente e robusto, por garantir que essas aplicações se comuniquem com um acordo preestabelecido. A estrutura do corpo da mensagem ou esquema deve ser acordada e comunicada da mesma forma que uma especificação de API. Esses esquemas precisam ser gerenciados e versionados para que possamos garantir a qualidade do fluxo de eventos.

Alguns *brokers* de eventos contemplam funcionalidades de gerenciamento de esquema e, normalmente, essas funcionalidades não são habilitadas por padrão. A forma como cada *broker* gerencia os esquemas pode variar. Vamos explorar como é o gerenciamento do esquema no Apache Kafka, da Confluent, uma empresa privada cujos fundadores desenvolveram o Apache Kafka no LinkedIn, uma das pioneiras no gerenciamento do esquema (https://www.confluent.io/). As formas como outros *brokers* lidam com o gerenciamento de esquemas são similares ou idênticas.

Vamos explorar três funcionalidades relacionadas ao gerenciamento de esquemas:

1. Registro de esquema (*Schema registry*);
2. Evolução ou versionamento de esquema (*Schema evolution* ou *Schema versioning*);
3. Verificação de compatibilidade de esquema (*Schema compatibility check*).

Registro de esquema

Segundo a definição da Confluent (c2023), um **registro de esquema** (*Schema registry*) é um serviço que armazena esquemas e suas diferentes versões à medida que evoluem. Produtores e consumidores recuperam os esquemas desse registro, os quais são armazenados localmente em seus caches e utilizados para serializar e desserializar o corpo dos eventos enviados e recebidos pelo *broker*, no caso, o Apache Kafka. Esse processo de serialização e desserialização acaba sendo a validação da estrutura do corpo dos eventos.

O registro de esquema é o componente principal de um *broker* relacionado ao gerenciamento de esquema. Ele permite o registro, a atualização, a visualização e o download dos esquemas. **Ele pode ser usado em tempo de design para descobrir esquemas disponíveis e em tempo de execução para as aplicações validarem a estrutura do corpo dos eventos.** Essas aplicações trocam informações sem ter que gerenciar e compartilhar o esquema.

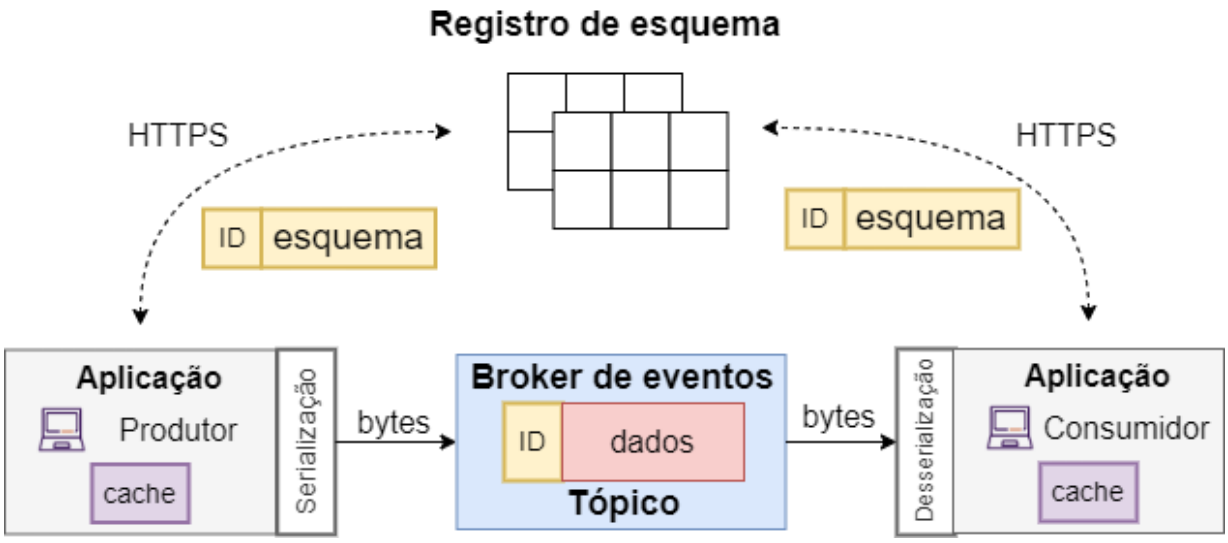

Figura 5.4: Fluxo de publicação e consumo de eventos com registro de esquema.

1. A aplicação produtora tenta serializar o corpo do evento chamando a biblioteca de serialização (serializador) fornecida pelo *broker* de acordo com o formato utilizado (e suportado pelo *broker*);
2. Isso aciona o serializador a verificar em seu cache local se o arquivo de esquema está disponível, mas não o encontra;
3. O serializador então consulta o registro de esquema via HTTPS e recupera o arquivo de esquema e o seu ID e os armazenam em seu cache local;
4. Com isso, o serializador serializa o corpo do evento, anexa o ID do esquema como parte do corpo do evento e envia para o *broker*;
5. A aplicação consumidora extrai o ID do esquema que recebeu do *broker* e verifica em seu cache local. Se o ID não estiver no cache, ela recupera o arquivo do esquema pelo ID no registro de esquema via HTTPS e o armazena em seu cache local;
6. Uma vez em posse do arquivo de esquema, ela desserializa o corpo do evento.

Há dois pontos que podemos destacar no fluxo. O primeiro é que, a menos que o ID do esquema seja alterado devido a uma evolução do esquema, os clientes se comunicam com o registro de esquema

apenas uma vez para recuperar o arquivo de esquema. O segundo ponto é que, como apenas o ID do esquema e os dados em si é que trafegam pelo *broker*, o tamanho do corpo do evento é menor por não precisarmos anexar toda a definição do esquema junto com o dado na publicação do evento.

Agora que sabemos como funciona o fluxo de publicação e consumo de eventos com um registro de esquema habilitado, vamos entender como esse serviço trabalha com evolução ou versionamento de esquema e a verificação de compatibilidade de esquema.

Evolução ou versionamento de esquema

Quando criamos nosso primeiro esquema, temos que ter uma coisa em mente: em algum momento, ele vai precisar evoluir. Com o tempo, a necessidade do negócio muda; com isso, precisamos atualizar nossas estruturas de dados ou esquemas de acordo para acompanhar essa evolução. A evolução ou versionamento é um aspecto fundamental no gerenciamento do esquema.

Quando isso acontece, é fundamental que as aplicações sejam capazes de interpretar os dados com o esquema antigo e o novo. Para garantir que essas aplicações não *quebrem* com essa evolução, o registro de esquema tem uma funcionalidade muito importante: a *verificação de compatibilidade de esquema*.

Verificação de compatibilidade de esquema

A **verificação de compatibilidade de esquema** valida as alterações dos esquemas no momento em que eles são registrados para garantir a compatibilidade entre versões, de acordo com o modo de compatibilidade configurado.

Pode ser feita na fase de desenvolvimento, *em design time*, e no momento da publicação e consumo de eventos, *em runtime*. Isso garante que as aplicações consumidoras continuem funcionando

com a evolução ou versionamento. Mas sua atualização é necessária para contemplar as alterações, caso seja do seu interesse.

Cada fornecedor de *broker* define o escopo da verificação. Na Confluent, é feita por *subject*. Outros fornecedores podem ter nomes diferentes, mas a ideia é a mesma — por exemplo, no Azure Event Hubs o nome é *grupo de esquemas*.

> **SUBJECT**
>
> *Subject* é um agrupador lógico de esquemas e suas versões no qual a verificação de compatibilidade é executada (https://developer.confluent.io/learn-kafka/schema-registry/schema-subjects/).

O modo de compatibilidade determina como o registro de esquema valida o novo esquema com a versão anterior para um determinado *subject*. Precisamos definir um modo quando registramos um esquema (caso contrário, é utilizado o modo padrão/*default* do *broker*).

Quando um esquema é criado para um *subject*, ele ganha um ID único e uma número de versão (exemplo: versão 1). Quando o esquema é atualizado (passa pela verificação de compatibilidade), ele ganha um novo ID único e uma nova versão incrementada (exemplo: versão 2).

Vamos explorar os modos **compatibilidade com versão anterior** (*backward compatibility*) e **compatibilidade com versão futura** (*forward compatibility*) e suas extensões, chamadas de *modos transitivos* (*transitive modes*). Para explorar melhor os modos, vamos usar um esquema de *Pedido* de exemplo com o formato binário Apache Avro. O esquema original que registramos no registro de esquema tem três campos obrigatórios: duas *strings* e um *double*. Essa é a versão 1 do nosso esquema de *Pedido*.

```
{ "namespace": "com.mycorp.pedido",
  "type": "record",
  "name": "Pedido",
  "fields":
   [ {"name": "orderId","type": "string"},
     {"name": "totalAmount","type": "double"},
     {"name": "status","type": "string"}
   ]
}
```

Compatibilidade com versão anterior

A *compatibilidade com versão anterior* garante que aplicações consumidoras mais novas possam atualizar seus esquemas e ainda consumir eventos publicados pelas aplicações produtoras mais antigas.

Por exemplo, se a nova versão registrada do esquema for `v3` e a aplicação consumidora atualizar para a `v3`, ela será compatível com a versão anterior imediata, `v2`, mas não necessariamente com a versão `v1`.

As alterações permitidas nesse modo são:

1. **Exclusão de campos**: aplicações produtoras antigas podem continuar a incluir este campo; novas aplicações consumidoras vão ignorá-lo;
2. **Adição de campos opcionais com um valor *default***: aplicações produtoras antigas não incluem esse campo; novas aplicações consumidoras usam o valor *default*.

A seguir, um exemplo válido de uma evolução do esquema de *Pedido* com o modo de *compatibilidade com versão anterior* (passou pela verificação de compatibilidade) configurada. O campo *status* foi removido.

```
{ "namespace": "com.mycorp.pedido",
  "type": "record",
  "name": "Pedido",
```

```
  "fields":
   [ {"name": "orderId","type": "string"},
     {"name": "totalAmount","type": "double"}
   ]
}
```

Compatibilidade com versão futura

A *compatibilidade com versão futura* garante que as aplicações produtoras mais novas possam publicar eventos com o esquema atualizado, que ainda pode ser consumido pelas aplicações consumidoras mais antigas.

Por exemplo, se a nova versão registrada do esquema for `v3`, as aplicações consumidoras serão compatíveis com a versão `v3`, mesmo ainda usando a versão `v2`.

As alterações permitidas nesse modo são:

1. **Adição de campos**: novas aplicações produtoras incluem este campo; aplicações consumidoras antigas o ignoram;
2. **Exclusão de campos opcionais com um valor *default***: novas aplicações produtoras não incluem esse campo; aplicações consumidoras antigas usam o valor *default*.

A seguir, um exemplo válido de uma evolução do esquema de *Pedido* com o modo de *compatibilidade com versão futura* (passou pela verificação de compatibilidade). O campo *address* foi adicionado.

```
{ "namespace": "com.mycorp.pedido",
  "type": "record",
  "name": "Pedido",
  "fields":
   [ {"name": "orderId","type": "string"},
     {"name": "totalAmount","type": "double"},
     {"name": "status","type": "string"},
     {"name": "address","type": "string"}
   ]
}
```

Temos uma terceira opção, chamada **FULL**, que é a combinação dos dois modos vistos anteriormente.

Por padrão, a verificação de compatibilidade que vimos verifica apenas a versão anterior imediata do esquema que estamos tentando registrar no registro de esquemas. Podemos estender a verificação de compatibilidade do esquema para todas as versões anteriores à que estamos registrando, e não somente à versão anterior imediata no contexto do *subject*. Essa extensão são os modos *transitivo retrógrado* (*backward transitive*), *transitivo direto* (*forward transitive*) e *transitivo completo* (*full transitive*).

A Confluent disponibiliza uma tabela interessante em sua documentação com o resumo dos modos de compatibilidade e suas extensões, que você pode encontrar em: https://docs.confluent.io/platform/current/schema-registry/avro.html#compatibility-types.

Até então, vimos a abordagem de evoluir ou versionar o esquema para um determinado tópico. Nessa abordagem, o tópico pode conter eventos que usam novos e antigos esquemas:

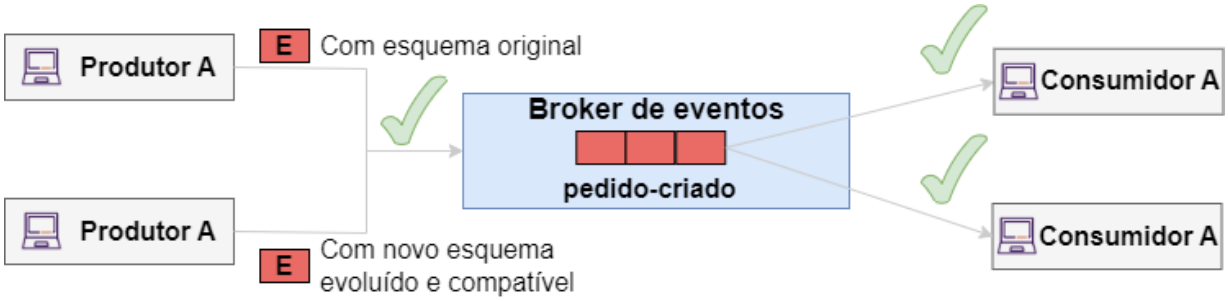

Figura 5.5: Único tópico com diversas versões compatíveis.

Uma segunda abordagem de versionamento

Uma outra abordagem que podemos utilizar é, além de evoluir ou versionar o esquema, também versionar o tópico. Nesse abordagem, em vez de termos um único tópico com diversas versões de

esquemas, podemos ter um tópico por versão. Por exemplo, *pedido-criado-v1*, *pedido-criado-v2*, e assim por diante.

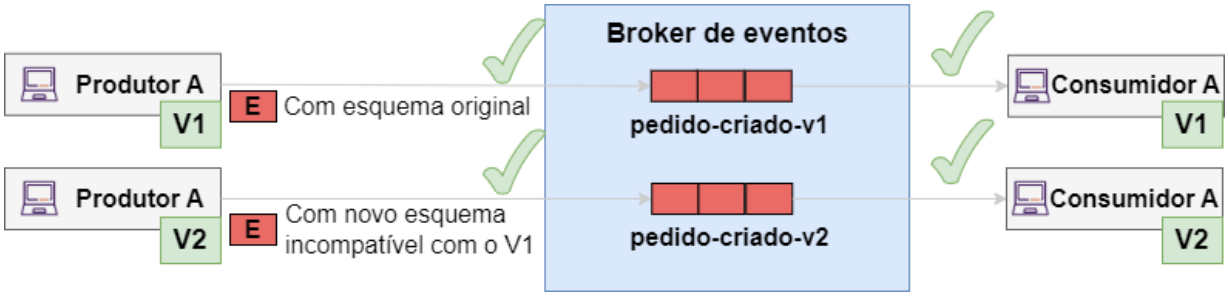

Figura 5.6: Um tópico por nova versão que quebra contrato.

Como regra geral, usamos a primeira abordagem quando for possível ter alterações no esquema que não *quebrem* o contrato. O máximo que pode acontecer é o consumidor não ser atualizado contemplando a alteração da nova versão, mas não precisa fazer um novo *deploy* dele para o fluxo funcionar. Nessa abordagem, não necessariamente precisamos versionar as aplicações por não ter quebra de contrato.

A segunda abordagem é ideal quando precisamos fazer alterações no novo esquema que são incompatíveis com os esquemas anteriores (não conseguimos seguir as regras de adição e remoção de campos de cada modo de compatibilidade). Por se tratar de alteração com quebra de contrato, possivelmente precisaremos versionar não somente o tópico, mas também as aplicações/microsserviços produtoras e consumidoras. Quando as aplicações consumidoras terminarem de migrar para o tópico `v2`, podemos aposentar o tópico `v1` e as aplicações `v1` da nossa solução.

Concluindo

Entendermos como o *broker* de eventos habilita a EDA é fundamental, pois o *broker* é considerado o coração da solução. Neste capítulo, entendemos o papel do *broker* na EDA e vimos a importância da escolha do padrão de implantação do *broker*. Vimos

também a relevância que as capacidades de operação e governança têm no processo de escolha do *broker*. Por fim, vimos o valor de termos as capacidades do gerenciamento de esquema no fluxo de publicação e consumo de eventos.

No próximo capítulo, vamos explorar outras funcionalidades do *broker*, estas mais voltadas para a parte de execução.

CAPÍTULO 6
Funcionalidades comuns de um broker no fluxo de publicação e consumo de eventos

No capítulo anterior, vimos a relevância do tipo de implantação, das capacidades da operação e governança, bem como do gerenciamento de esquemas no processo de escolha do *broker*. Neste capítulo, vamos explorar algumas das principais capacidades dos *brokers* de eventos relacionadas ao fluxo de publicação e consumo dos eventos.

Como vimos na seção *Componentes-chaves* do capítulo 2, *Entendendo os principais conceitos*, as aplicações interagem com os *brokers* como produtoras e/ou consumidoras de eventos. As aplicações no papel de produtoras publicam eventos/mensagens e podem esperar uma confirmação (*acknowledgement*, ou *ack*) de recebimento do *broker*. Essa interação pode ser síncrona (chamada REST API), *request-response* ou assíncrona.

Já as aplicações consumidoras **podem interagir com o *broker* com base no mecanismo de entrega *push* ou *pull***. Elas podem confirmar o recebimento do evento/mensagem ao *broker*. Vamos falar sobre a diferença do mecanismo de *push* e *pull* e da *confirmação de recebimento* na perspectiva da aplicação produtora e consumidora em breve neste capítulo.

Entender como as aplicações dentro da sua solução de EDA interagem com o *broker*, bem como as capacidades do *broker* referentes ao fluxo de publicação e consumo de eventos, é de extrema importância no processo de escolha do fornecedor do *broker*. Você deve estar ciente dessas capacidades e formas de interação para garantir que sua solução se comporte conforme o esperado.

6.1 Interação da aplicação consumidora com base em push e pull

As aplicações consumidoras podem interagir com o *broker* para consumir um evento de um determinado tópico utilizando dois mecanismos de entrega: *push* ou *pull*. Vamos explorar os dois mecanismos e entender os casos de uso que são mais apropriados para cada um deles.

Push

Nesse mecanismo de entrega, a comunicação é iniciada pelo *broker* de eventos: é ele que envia – ou "empurra" (*push*) – o evento para a aplicação consumidora registrada.

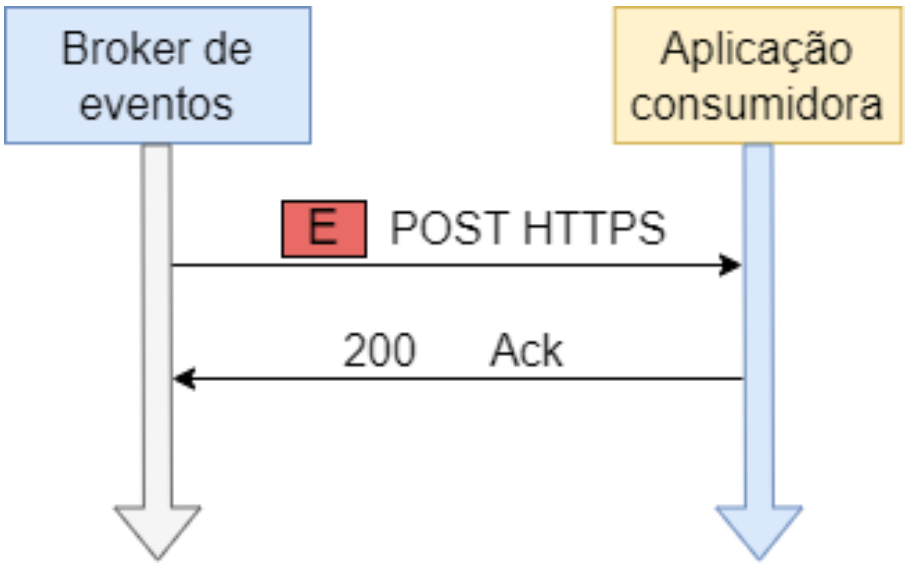

Figura 6.1: Mecanismo de entrega de eventos push.

Uma forma comum de entrega de evento desse mecanismo é utilizar um *endpoint* `HTTP(S)` público da aplicação consumidora registrada no tópico. Esse *endpoint* normalmente é um `POST`. A aplicação consumidora recebe a notificação do *broker* e retorna um *status code* HTTP de sucesso ou de erro, confirmando o recebimento do evento. O *broker* de eventos pode oferecer políticas de retentativas

com base em erros transitórios ou de acordo com a família do *status code HTTP* — `4xx`, `5xx`.

Outras formas de entrega de eventos são possíveis, como acionar funções *serverless* ou serviços diversos. Por exemplo, o *broker* AWS EventBridge pode notificar através do mecanismo de entrega *push* um AWS Lambda, um tópico Amazon Simple Notification Service (SNS) ou outros serviços da própria AWS. O mesmo pode ser feito com o *broker* Google Cloud Pub/Sub: ele pode notificar outros serviços do Google Cloud.

O mecanismo de entrega *push* é mais indicado nos casos onde precisamos utilizar o padrão *webhook*. Além disso, ele é ideal para aplicações consumidoras nativas da nuvem, que têm essa possibilidade de comunicação, como o exemplo do AWS EventBridge e AWS Lambda mencionado anteriormente. Esse mecanismo não é indicado quando precisamos processar um grande volume de eventos em tempo real (normalmente fora da realidade da maioria das empresas, algo em torno de GB por segundo) e onde a eficiência e o *throughput* são críticos.

Pull

Nesse mecanismo de entrega, a comunicação é iniciada pela aplicação consumidora. A aplicação consumidora consome – ou "puxa" (*pull*) – os eventos do *broker*.

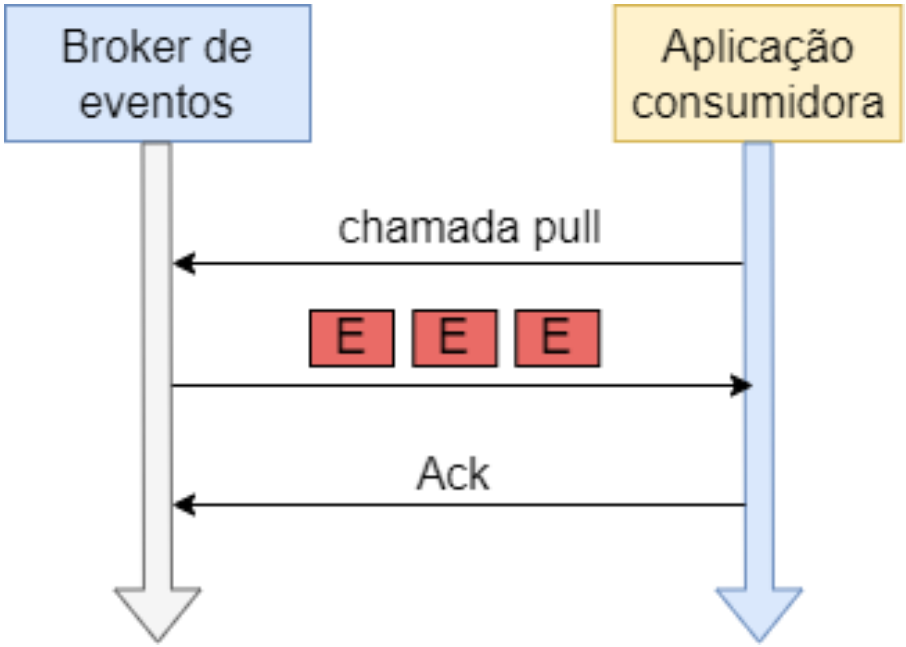

Figura 6.2: Mecanismo de entrega de eventos pull.

O mecanismo de entrega *pull* é mais indicado nos casos de uso em que precisamos processar um grande volume de eventos em tempo real com padrão de entrega de eventos *event streaming*, como vimos na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*.

Para processar um grande volume de eventos em tempo real, esse mecanismo utiliza uma comunicação bidirecional, que mantém uma conexão aberta com o *broker*, iniciada pela aplicação consumidora, como o protocolo Websocket. Alguns *brokers* utilizam a estratégia de *polling* no mecanismo de *pull*. Esses fornecedores dão as opções de *polling curto* ou *polling longo*, como o AWS SQS.

O protocolo de mensagem suportado pelo *broker* que você escolher é que determina o tipo de mecanismo utilizado. Um protocolo de mensagem do tipo `HTTP(S)` fica evidente que é *push* (iniciado pelo *broker*), mas existem outros menos evidentes, como *AMQP 0.9.1* e *AMQP 1.0*. As bibliotecas ou SDKs, que veremos a seguir, geralmente ocultam esse mecanismo de entrega, abstraindo toda a complexidade do protocolo.

6.2 Kit de desenvolvimento de software (SDK)

Através do **Kit de desenvolvimento de software** (do inglês *Software Development Kit*, ou simplesmente SDK), fornecedores de *brokers* podem oferecer meios para abstrair toda a complexidade do protocolo de comunicação nas aplicações ou microsserviços que publicam e consomem eventos. Alguns fornecedores utilizam outras nomenclaturas, como *APIs de mensagens* ou *bibliotecas do cliente*. Vamos utilizar a sigla *SDK* neste livro.

O SDK é mais uma opção de interação com o *broker*, como vimos na seção *Operação e governança* do capítulo 5, *Broker de eventos habilita a arquitetura orientada a eventos*. Apesar de alguns *brokers* permitirem fazermos configurações globais em nível de tópico e *broker* via SDK, ele é idealmente usado para fazermos configurações no nível de aplicações produtoras e consumidoras, que alteram o comportamento das aplicações no fluxo de publicação e consumo de eventos.

Por exemplo, ao publicar um evento, podemos configurar o tempo máximo que a aplicação produtora aguardará pela confirmação de recebimento do *broker*. Essas configurações são normalmente definidas na inicialização das aplicações.

Os SDKs podem ser oferecidos em diversas linguagens de programação, como Java, Javascript/Node.js e Python, para citar algumas.

Vamos ver um exemplo de uma publicação de eventos assíncrona utilizando o SDK Java, da Confluent (https://docs.confluent.io/kafka-clients/java/current/overview.html). Por questão de simplicidade, omiti o trecho da parte da inicialização e da aplicação consumidora para consumir o evento.

```
final ProducerRecord<K, V> record = new ProducerRecord<>(topic, key,
value);
```

```
producer.send(record, new Callback() {
  public void onCompletion(RecordMetadata metadata, Exception e) {
    if (e != null)
      log.debug("Send failed for record {}", record, e);
  }
});
```

Vamos ver um outro exemplo de uma publicação de eventos assíncrona, agora utilizando o SDK Python, da Confluent (https://docs.confluent.io/kafka-clients/python/current/overview.html). Por questão de simplicidade, também omiti o trecho da parte da inicialização e da aplicação consumidora para consumir o evento:

```
def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (str(msg)))

producer.produce(topic, key="key", value="value", callback=acked)
```

No momento do processo de escolha do fornecedor do *broker* para nossa solução de EDA, precisamos saber se ele oferece algum SDK para ganharmos agilidade no desenvolvimento e em quais linguagens de programação. Isso pode determinar o *broker* da solução ou o perfil do time para o projeto.

> Muitas das capacidades que veremos a partir de agora têm seus valores *default* definidos no *broker* e atenderão boa parte dos casos de uso sem precisarmos alterá-las. Mas é bom conhecermos essas capacidades e suas configurações *default* para nos auxiliar em possíveis necessidades de negócio, otimização e resolução de problemas no *broker*.

6.3 Envio e recebimento de eventos em lote

Por padrão, quando falamos no fluxo de publicação e consumo de eventos, estamos falando em eventos publicados e consumidos de forma individual, seja de um evento discreto ou evento de série (*stream*). Mas o *broker* pode permitir que as aplicações produtoras e consumidoras publiquem e consumam os eventos em lote (*batch*) para minimizar o tráfego de rede na comunicação entre as aplicações e o *broker*.

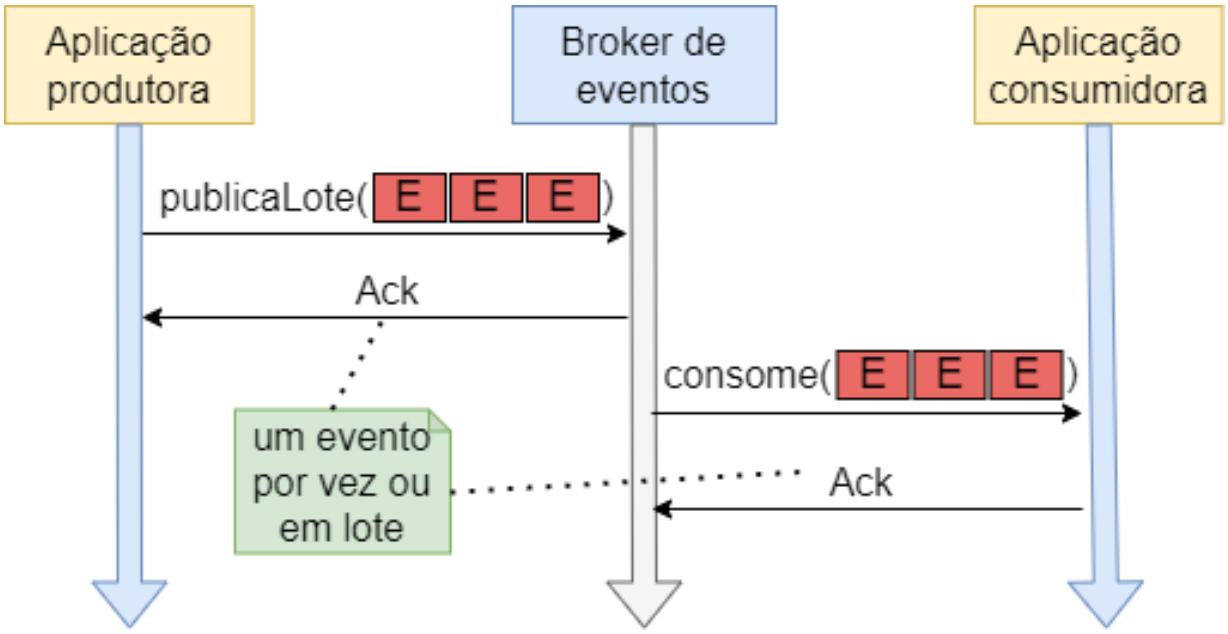

Figura 6.3: Envio e recebimento de eventos em lote.

A configuração do lote pode ser feita com base na quantidade de eventos ou em *bytes* pelo *broker*. Por exemplo, na perspectiva da aplicação produtora, podemos configurar um limite de 100KB em cada publicação/envio para o *broker*. A quantidade de eventos vai depender do seu tamanho (todo *broker* tem um valor *default* máximo do tamanho do evento (*payload*); no Apache Kafka, por exemplo, é 1MB).

Outro fator importante é o tempo que o *broker* aguarda para publicar os eventos em lote. Essa configuração é o tempo que a

aplicação espera para acumular em *bytes* para enviar para o *broker* respeitando o limite configurado. Por exemplo, se configurarmos 10 milissegundos e isso em média equivaler a 20 eventos ou 94KB, isso significa que evitamos 19 publicações/envios a mais ao *broker* nesse período.

O efeito colateral é que essa configuração adiciona até 10 milissegundos de atraso para enviar os eventos em lote e a quantidade pode estar bem abaixo do limite configurado, mas ainda ganhamos ao reduzir o número de interação com o *broker*.

Na perspectiva da aplicação consumidora, a quantidade em *bytes* ou a quantidade de eventos configurada para buscar (*fetch*) os eventos em lote tem que estar de acordo com sua capacidade de processamento. Colocar um valor pequeno pode não gerar benefícios e colocar um valor muito grande pode sobrecarregá-la. Caso seja necessário alterar para possíveis otimizações, é uma boa prática fazer teste de carga simulando essas configurações para chegar em um número que gere benefícios/desempenho.

Pré-busca (prefetch)

Os *brokers* de eventos que utilizam o mecanismo de entrega *push*, como RabbitMQ e ActiveMQ (padrão da versão do seu protocolo AMQP — essa complexidade é ocultada pelo SDK), podem oferecer funcionalidade de *pré-busca* (*prefetch*). A configuração da pré-busca é limitar o número máximo de eventos que podem ser enviados/empurrados para um único consumidor de uma só vez (*batch*).

A aplicação consumidora armazena esses eventos no seu cache local ou *buffer* até que possa processá-los. Ela precisa dimensionar seu cache local de acordo com o limite da pré-busca configurada. Um valor alto na configuração do limite da pré-busca é recomendado quando tivermos uma única ou poucas aplicações consumidoras processando os eventos rapidamente. Isso faz com que as aplicações fiquem o mais ocupadas possível. Em uma situação onde

temos muitas aplicações consumidoras com tempo de processamento curto, a recomendação é termos um valor pequeno na configuração do limite da *pré-busca*.

Já se tivermos muitas aplicações no mesmo grupo de consumidores e/ou tempo de processamento longo, a recomendação é desativar essa configuração ou, dependendo do *broker*, colocar com valor equivalente a um evento por vez na pré-busca. Isso faz com que os eventos consumidos no *broker* sejam distribuídos uniformemente entre todas as aplicações consumidoras do mesmo grupo.

A publicação e consumo de eventos em lote é ideal para casos de uso em que temos um alto volume de eventos no nosso fluxo para minimizar o impacto do excesso de requisições (ida e volta) ao *broker*.

6.4 Roteamento inteligente

Há casos onde precisamos fazer *multicast* dos eventos para apenas um conjunto de aplicações consumidoras registradas no mesmo canal (tópico ou fila) baseado em uma regra de filtro.

Suponha que nossa solução de *e-commerce* tenha dois tipos de pedido: *digital* e *físico*. A aplicação consumidora `fulfillment-digital` só sabe processar pedidos do tipo *digital*. Já a aplicação consumidora `fulfillment-fisico` , apenas pedidos do tipo *físico*. Nessa situação, naturalmente pensamos como primeira opção de solução fazer ambas aplicações consumidoras se registrarem no tópico *pedido-criado*, que contempla os dois tipos de pedido, e cada aplicação receberia uma cópia do pedido (*pub/sub*) e verificaria qual o tipo em questão. Dependendo do tipo, a aplicação processaria ou descartaria o evento. Um grande problema nessa solução é o processamento desnecessário das aplicações com descarte de eventos que não correspondem ao tipo esperado.

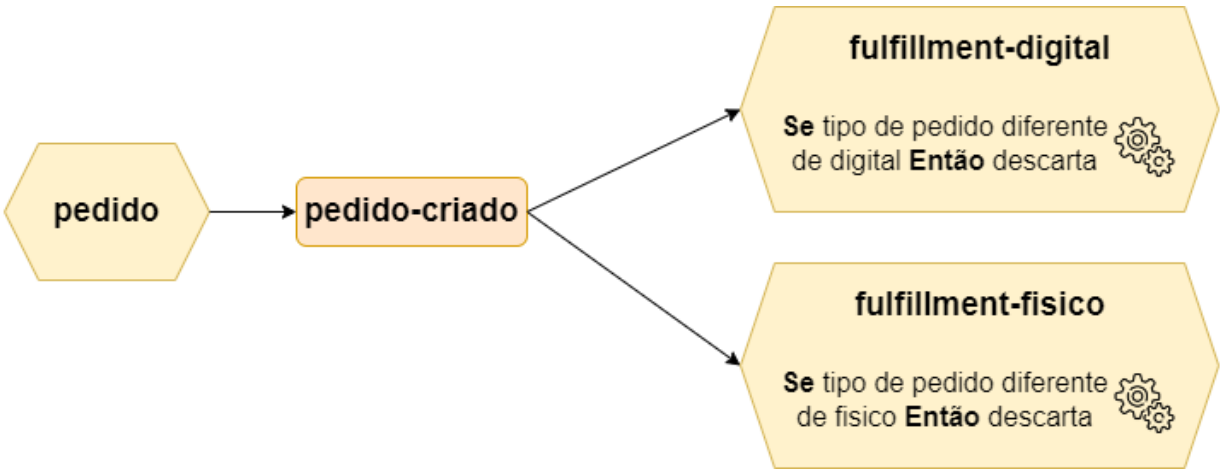

Figura 6.4: Solução sem roteamento inteligente com processamento de descarte.

Uma segunda opção seria criarmos dois tópicos, *pedido-criado-digital* e *pedido-criado-fisico* e cada aplicação consumidora ficaria registrada nos tópicos equivalentes. A aplicação produtora ficaria com a responsabilidade de interpretar o tipo e publicar em um dos tópicos. E se surgisse um novo tipo de pedido? Aí começa o problema dessa opção: a cada novo tipo, precisaríamos alterar a aplicação produtora para adicionar mais uma condição de acordo com o tipo e publicar em um novo tópico.

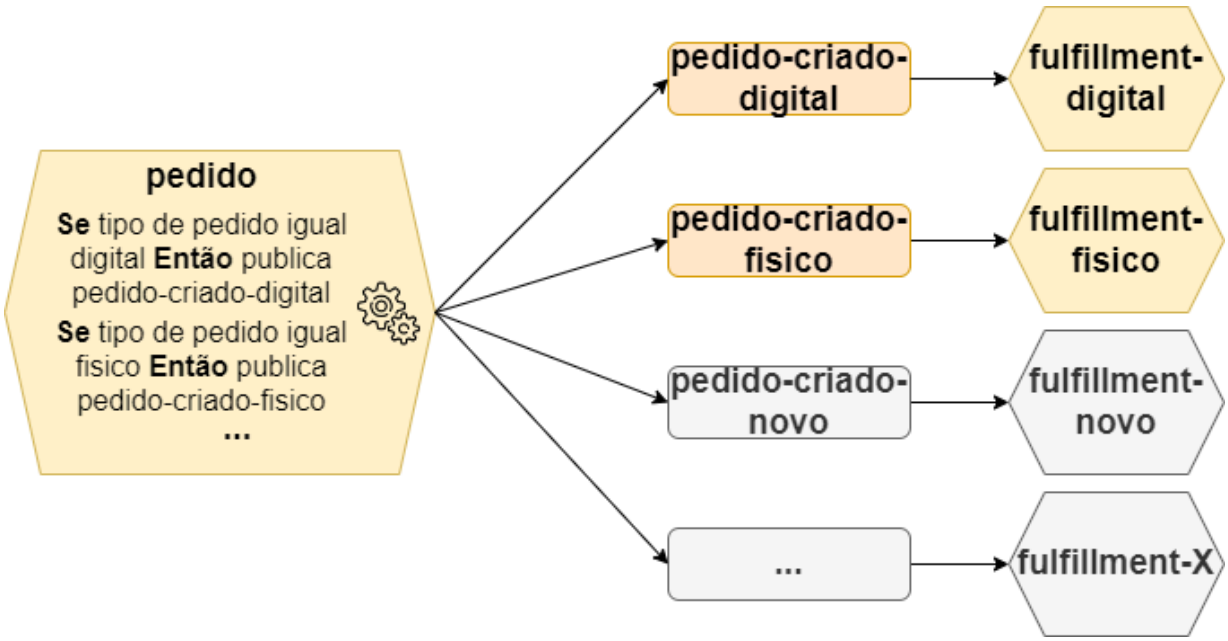

Figura 6.5: Solução sem roteamento inteligente com split de evento por tipo no produtor

Caso o fornecedor do *broker* tenha a funcionalidade de roteamento inteligente, poderíamos ter um único tópico *pedido-criado* com as duas aplicações consumidoras registradas. Criaríamos uma regra de filtro para cada aplicação consumidora que consumiria apenas o evento de acordo com o filtro, nesse caso, pedido *digital* ou *físico*.

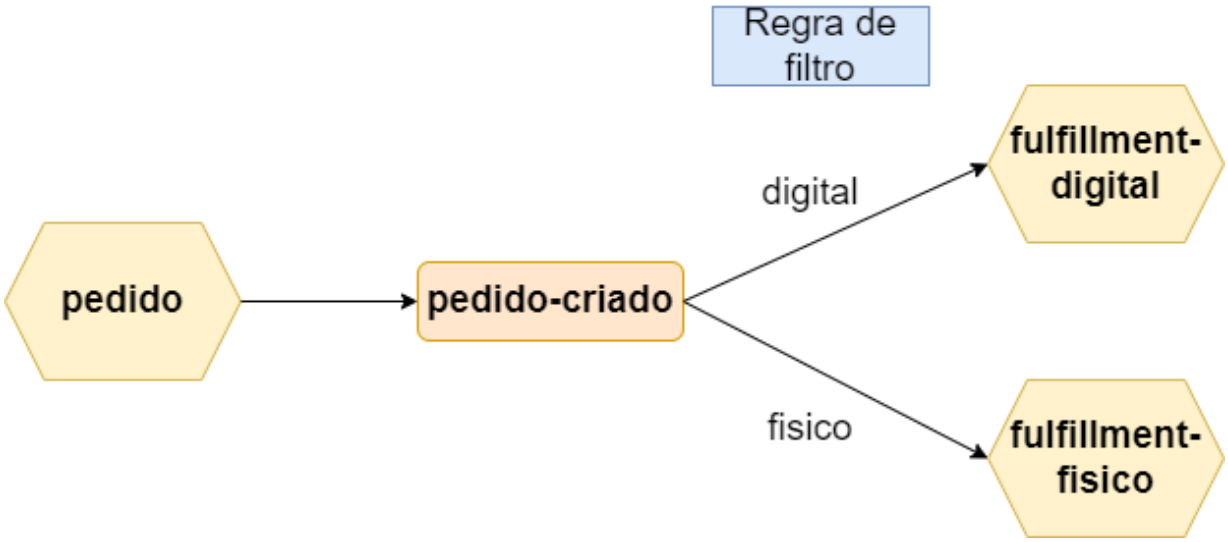

Figura 6.6: Solução com roteamento inteligente.

Com o roteamento inteligente, não teríamos desperdício de processamento para descarte de evento e, para cada novo tipo de pedido adicionado, a aplicação produtora poderia apenas informar o tipo no momento da publicação e criaríamos uma outra aplicação consumidora com a regra de filtro para o novo tipo.

Dependendo do fornecedor do *broker*, podemos usar *texto fixo*, *wildcards*, *condicional* ou *expressão regular* para criar as regras de filtro. Essas regras podem ser baseadas no nome do canal ou no conteúdo do corpo ou *header* do evento (*content-based*).

Podemos citar alguns exemplos de *brokers* que têm a funcionalidade de roteamento inteligente (alguns podem ter outros nomes, mas a ideia é a mesma):

- Na Solace, a regra é baseada no nome do tópico (criado hierarquicamente separado com barra – A/B/C) utilizando *wildcards* (https://docs.solace.com/Messaging/Wildcard-Charaters-Topic-Subs.htm);
- No RabbitMQ, a regra pode ser baseada com *texto fixo* (https://www.rabbitmq.com/tutorials/tutorial-four-python.html) ou com *wildcards* (https://www.rabbitmq.com/tutorials/tutorial-five-python.html), ambas baseadas em *routing key* e *binding key*;
- Para finalizar, no Azure Event Grid (https://learn.microsoft.com/pt-br/azure/event-grid/event-filtering) e Amazon EventBridge (https://docs.aws.amazon.com/pt_br/eventbridge/latest/userguide/eb-event-patterns-content-based-filtering.html), as regras são baseadas nos valores dos campos do corpo do evento utilizando operadores de comparação.

6.5 Confirmação de entrega de eventos (Ack)

No capítulo 2, *Entendendo os principais conceitos*, vimos a importância da confirmação de recebimento de eventos no fluxo de publicação e consumo e das semânticas para a garantia de entrega. Nesta seção, vamos explorar os modos de confirmação de recebimento dos eventos na perspectiva da aplicação produtora e consumidora dos eventos. A configuração desses modos, junto a outras configurações oferecidas pelos *brokers*, vai determinar o nível da garantia de entrega para a nossa solução.

Aplicação produtora

Na perspectiva da aplicação produtora, podemos ter duas configurações de comportamentos quando ela publica um evento para o *broker*:

1. Ela não espera qualquer confirmação do *broker* de recebimento. O *broker* pode nem persistir o evento. Padrão conhecido como *fire-and-forget*;
2. Ela aguarda uma confirmação de recebimento do *broker* após ele persistir o evento, para assegurar que este não será perdido em caso de queda de rede ou reinicialização do *broker*.

A primeira configuração é mais indicada para uma semântica de *entrega no máximo uma vez* e para quando precisamos de um alto *throughput* e muito baixa latência de rede. A aplicação assume uma confirmação automática. Esse é um cenário onde é tolerável a perda de eventos. A segunda configuração é mais indicada para casos de uso onde a perda de eventos não é tolerada.

A aplicação pode aguardar a confirmação de forma síncrona ou assíncrona e ela pode ser positiva ou negativa. A confirmação positiva significa que o *broker* recebeu (persistiu) com sucesso o evento. Já a confirmação negativa pode ter vários motivos, como o

tópico não existir, a aplicação não ter permissão para publicar no tópico e por aí vai.

Podemos configurar o tempo de espera da confirmação do *broker*. Por exemplo, se o *broker* não responder em X milissegundos a confirmação, a aplicação produtora pode retentar, caso esteja habilitada uma política de retentativa no *broker*. Isso pode ocasionar eventos duplicados. Se a aplicação for configurada para não esperar uma confirmação, a configuração de retentativa fica sem efeito.

Para envio e recebimento de eventos em lote, o *broker* e a aplicação consumidora podem confirmar o recebimento dos eventos em lote.

Para citar alguns fornecedores, na Solace, utilizamos o padrão *fire-and-forget* com *Direct Messages* (https://docs.solace.com/Messaging/Direct-Msg/Direct-Messages.htm). No RabbitMQ, podemos habilitar o modo de confirmação do publicador no canal (https://www.rabbitmq.com/confirms.html#publisher-confirms). Já no Apache Kafka, o cliente consegue controlar o nível de garantia de entrega na perspectiva da aplicação produtora configurando o *ack* para sincronizar (persistir) o evento/dado em todas as partições/réplicas do tópico (https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#acks).

Aplicação consumidora

Na perspectiva da aplicação consumidora, ela precisa confirmar para o *broker* que o evento foi processado (ou pelo menos recebido). Temos dois modos de confirmação de recebimento: *automática* ou *manual*.

No modo de confirmação automática (normalmente a configuração *default* dos *brokers*), um evento é considerado entregue com sucesso imediatamente após o envio para a aplicação consumidora. Isso faz com que o evento não fique disponível para execução para

outros consumidores do mesmo grupo; ou, dependendo do tipo do *broker*, o evento pode ser removido, pois foi considerado *entregue*. O problema que pode ocorrer é se o processamento da aplicação consumidora falhar: como o evento já foi confirmado automaticamente, o evento pode não ser processado ou ser perdido.

A confirmação automática é recomendada para processamentos curtos, onde ganhamos eficiência no fluxo e simplicidade no desenvolvimento da aplicação por não se preocupar com confirmação manual via código.

No modo de confirmação manual, a aplicação consumidora confirma o recebimento programaticamente através do SDK, normalmente após o processamento do evento de negócio. Caso a aplicação consumidora não consiga finalizar o processamento com sucesso, por causa de alguma exceção, ou não o confirme dentro de um período de tempo definido (*broker* que oferece a funcionalidade de visibilidade do evento), o evento pode ser processado novamente por outra aplicação do mesmo grupo. Isso pode gerar eventos duplicados.

A confirmação manual é recomendada para processamento mais longos, onde queremos ter um controle maior e garantir que o evento seja confirmado ou removido, dependendo do *broker*, apenas após o processamento com sucesso.

Na Solace, utilizamos o *Guaranteed Messages* para garantir a entrega dos eventos com confirmação de recebimento (https://docs.solace.com/Messaging/Guaranteed-Msg/Publishing-Guaranteed-Messages.htm). No *RabbitMQ*, podemos confirmar o recebimento de forma automática ou programaticamente na aplicação consumidora de forma manual. A confirmação manual pode ser positiva ou negativa. A confirmação negativa pode fazer o evento ser removido, republicado na mesma fila de origem para ser reprocessado ou movido para uma DLQ configurada (https://www.rabbitmq.com/confirms.html#acknowledgement-modes). No Apache Kafka, também podemos confirmar o

recebimento (conhecido como registro de confirmação, ou *commit log*) de forma automática ou programaticamente na aplicação consumidora, de forma manual. Na forma manual, podemos confirmar de forma síncrona ou assíncrona, com os métodos `commitSync()` ou `commitAsync()`, respectivamente (https://docs.confluent.io/cloud/current/client-apps/optimizing/durability.html#consumer-offsets-and-auto-commit).

Essas configurações – tanto da perspectiva da aplicação produtora quanto da consumidora – determinam o nível de desempenho e de garantia de entrega que queremos para nossa solução de EDA de acordo com os requisitos de negócio.

6.6 Retenção do evento

A **retenção de eventos**, também conhecida como retenção de log (*log retention*), é uma capacidade do *broker* de manter o evento armazenado de acordo com a configuração de uma política de retenção. Apesar de a configuração em dias, horas ou minutos ser mais popular, podemos ter políticas de retenção em *bytes*.

O *broker* pode manter o evento armazenado por um período indeterminado ou definir um período máximo. O cliente pode definir um período dentro do limite estipulado pelo *broker*. O ponto-chave é que o evento será removido do tópico/fila para liberar espaço após o período de retenção.

Dependendo do tipo do *broker*, é usado o termo *expiração* ou TTL (*time-to-leave*). O objetivo é o mesmo: reter o evento por um determinado período.

Existe uma outra forma que o *broker* usa para remover o evento além da configuração da retenção ou TTL. Dependendo do tipo do *broker*, ele pode remover o evento após todas as aplicações consumidoras registradas confirmarem/processarem o evento.

Vamos ver esse tipo de comportamento em detalhe no próximo capítulo, quando falarmos dos tipos de *brokers*.

É importante avaliarmos a capacidade do *broker* de reter eventos no momento do processo de escolha. O caso de uso da nossa solução de EDA vai depender dessa capacidade do *broker*. Os *brokers* que têm a capacidade de armazenar eventos são considerados como *event store*.

Eventos de negócio tendem a ter seu valor degradado ao passar do tempo depois da publicação. Nesse caso, um evento ou tópico com período de retenção ou TTL curtos resolveria o problema. Por exemplo, apesar de o Apache Kafka nos permitir configurar um período indeterminado de retenção para os tópicos, sua configuração *default* é de 7 dias.

Para casos de uso em que precisamos processar os eventos/dados históricos, precisaríamos de um *broker* que permite reter eventos por um prazo indeterminado ou por um longo prazo. Esses tipos de *brokers* são baseados em logs, como veremos no capítulo seguinte. Por exemplo, um banco que queira proporcionar uma melhor experiência para seus clientes ao permitir que as pessoas busquem todas as suas transações, e não somente as dos últimos três meses.

> O *broker* nem sempre é o melhor lugar para armazenarmos os dados históricos, por causa do custo. Isso tem que ser avaliado em conjunto com possíveis integrações com armazenamento externo.

Também devemos avaliar as regulamentações da indústria em que o cliente está inserido. Algumas delas exigem um tempo mínimo dos dados armazenados, entre outras conformidades.

6.7 Reprodução do evento

Vimos que uma das estratégias do *broker* para não perder o evento e garantir a entrega para a aplicação consumidora é armazenar o evento em uma memória não volátil. Com isso, ele garante que o evento não seja perdido por uma reinicialização do *broker* ou qualquer falha que venha ocorrer. Mas existe um outro motivo para o *broker* armazenar os eventos: permitir que aplicações consumidoras façam a reprodução (*replay*) dos eventos.

A **reprodução de evento** é uma capacidade do *broker* que permite que ele reenvie eventos para as aplicações consumidoras novas ou existentes que as solicitarem, horas ou até mesmo dias depois que os eventos foram recebidos pela primeira vez pelo *broker*. O período máximo em que essas aplicações podem receber os eventos vai até o limite de retenção configurado.

Essa capacidade ajuda as aplicações/soluções a ficarem em um estado consistente, por permitir que reprocessem toda a cadeia de eventos em caso de falhas. Ela está diretamente associada à retenção de eventos de longo período. Os *brokers* que têm essas capacidades são *brokers* baseados em logs para atender ao padrão de entrega *event streaming*, que vimos na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*. Vamos falar sobre esse tipo de *broker* no próximo capítulo.

Os *brokers* normalmente suportam algumas políticas de reprodução. Essas políticas são referentes à posição do log de eventos onde as aplicações consumidoras vão iniciar o consumo dos eventos no momento que se registram no tópico:

1. Iniciar a partir dos novos eventos;
2. Iniciar a partir do primeiro evento;
3. Iniciar a partir do último evento;
4. Iniciar a partir de uma sequência;
5. Iniciar a partir de uma data e hora;

6. Iniciar a partir de uma duração de tempo.

Iniciar a partir dos novos eventos

A aplicação consumidora inicia o consumo dos eventos a partir dos novos eventos publicados após o seu registro no tópico. Essa normalmente é a política *default* adotada pelos *brokers*.

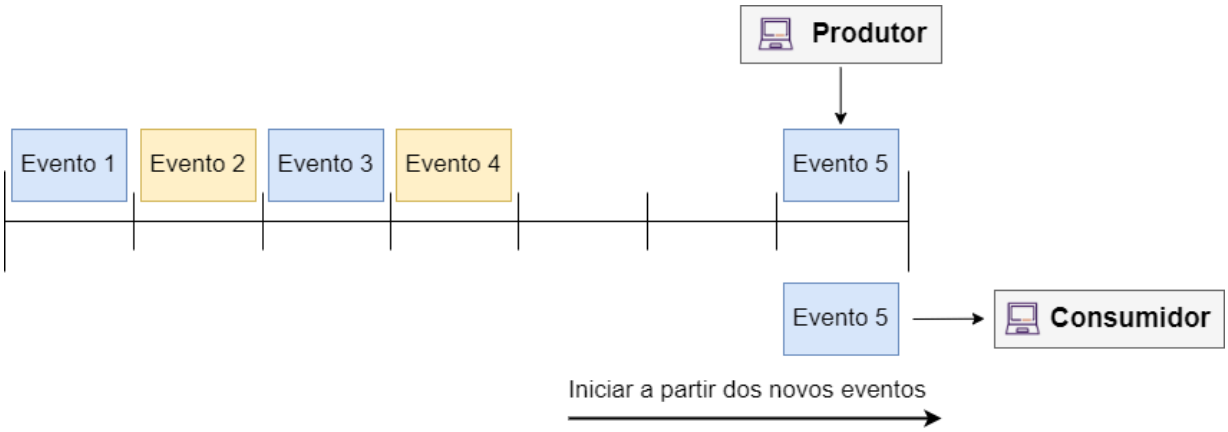

Figura 6.7: Reprodução de eventos – iniciar a partir dos novos eventos.

Iniciar a partir do primeiro evento

A aplicação consumidora inicia o consumo dos eventos a partir do primeiro evento do tópico. A quantidade de eventos a ser reproduzida vai depender do período de retenção configurado.

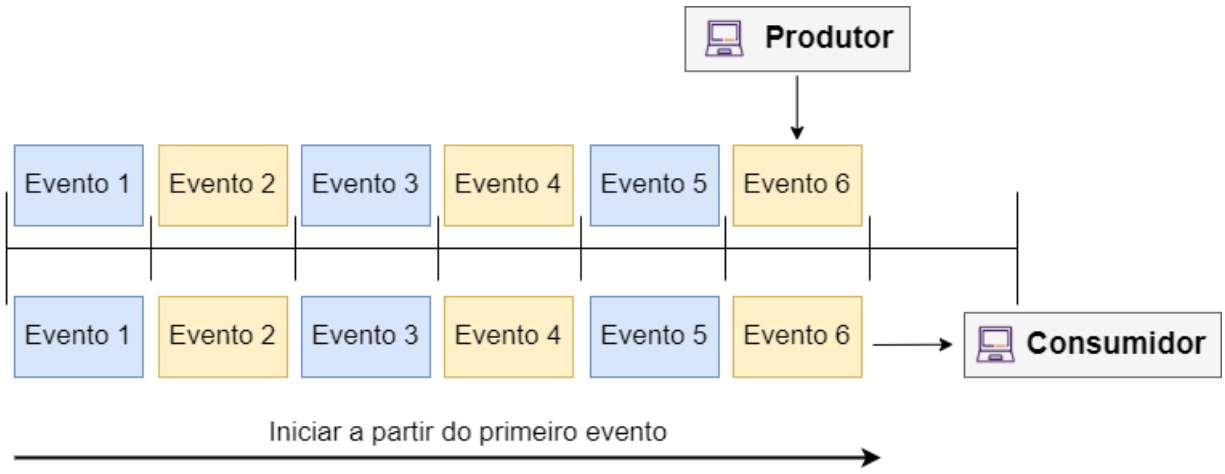

Figura 6.8: Reprodução de eventos – iniciar a partir do primeiro evento.

Iniciar a partir do último evento

A aplicação consumidora inicia o consumo dos eventos a partir do último evento publicado no tópico.

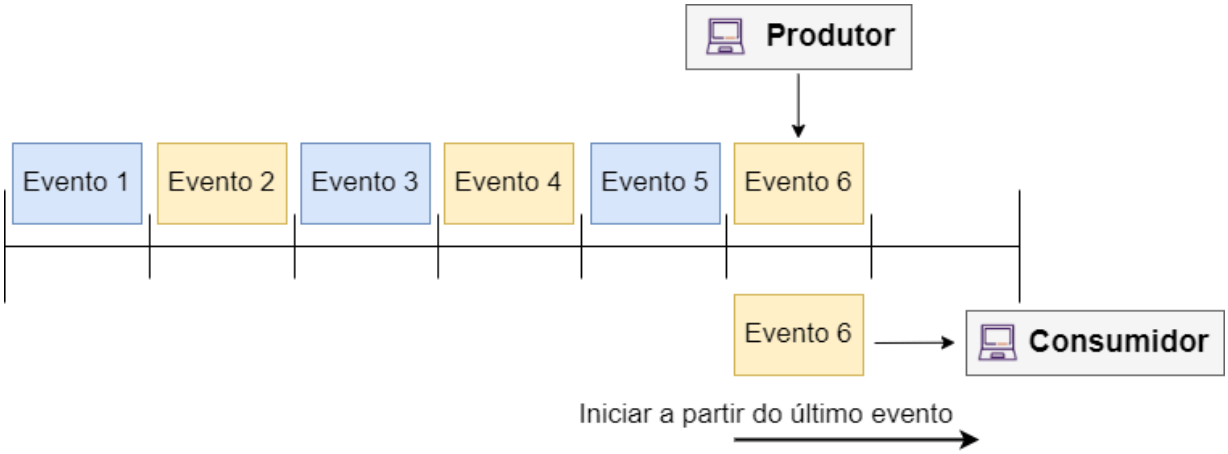

Figura 6.9: Reprodução de eventos – iniciar a partir do último evento.

Iniciar a partir de uma sequência

A aplicação consumidora inicia o consumo dos eventos a partir de um evento específico e segue a sequência dos eventos mais atuais.

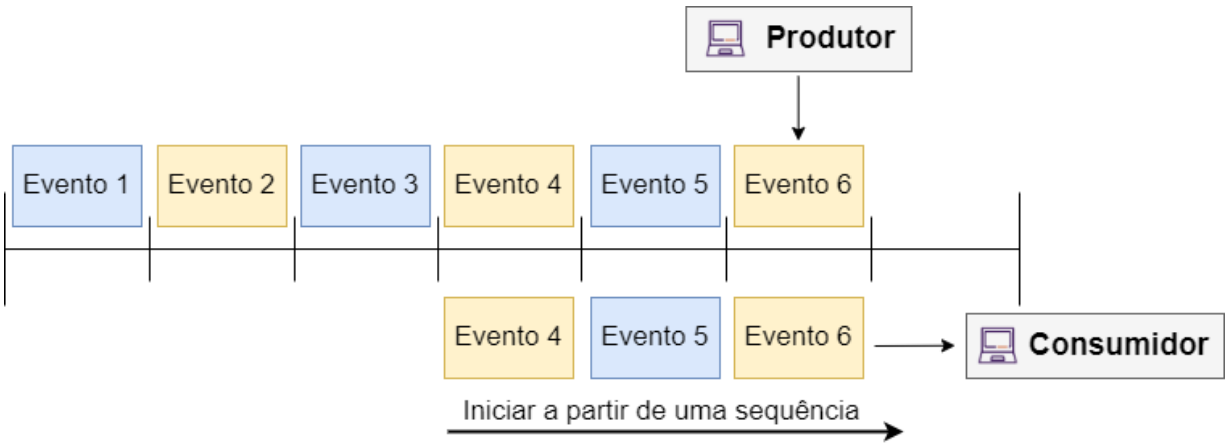

Figura 6.10: Reprodução de eventos – iniciar a partir de uma sequência.

Iniciar a partir de uma data e hora

A aplicação consumidora inicia o consumo dos eventos a partir de uma data e hora. Todos os eventos têm uma data e hora associadas a eles. A data e hora podem ser um ***event time***, que é a data e hora em que a aplicação produtora envia o evento para o *broker*, ou podem ser um ***ingestion time***, que é a data e hora do momento da gravação do evento no log de eventos.

O Apache Kafka nos permite definir qual data e hora queremos. A configuração *default* é o ***event time***. Em todas as soluções EDA que executei, não foi preciso alterar a configuração *default*, mas pode ocorrer de termos algumas necessidades específicas — e ter configurações flexíveis é sempre bom.

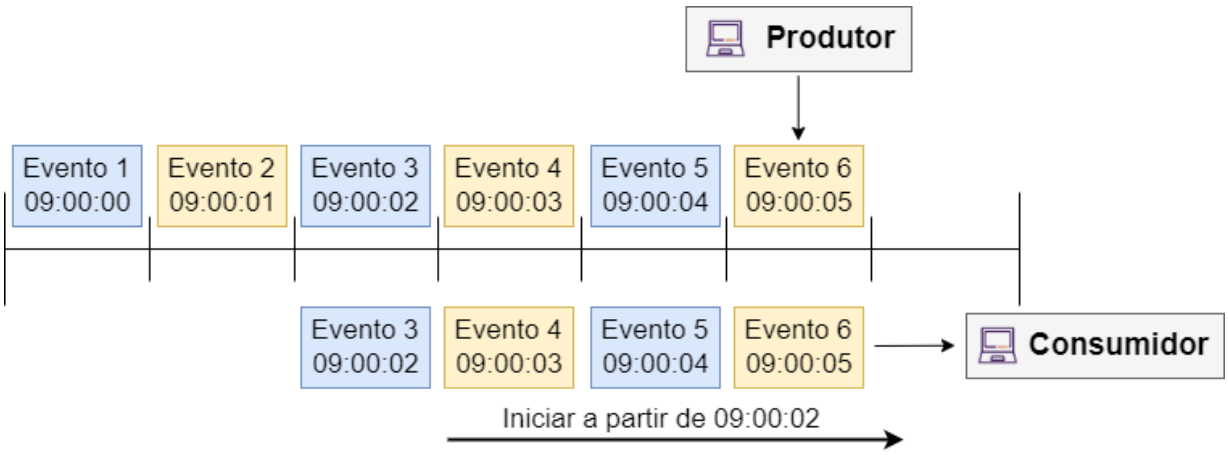

Figura 6.11: Reprodução de eventos – iniciar a partir de uma data e hora.

Iniciar a partir de uma duração de tempo

A aplicação consumidora inicia o consumo dos eventos a partir de uma duração.

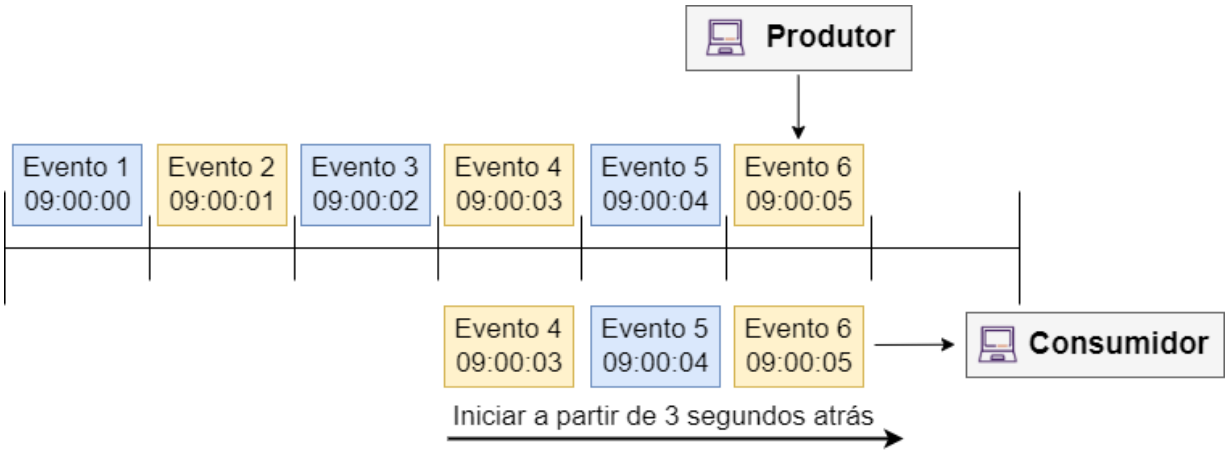

Figura 6.12: Reprodução de eventos – iniciar a partir de uma duração de tempo.

6.8 Visibilidade do evento

Imagine um cenário onde temos uma fila e três consumidores fazendo o balanceamento de carga — um cenário típico do padrão

de entrega *point-to-point*, como vimos na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*.

Como garantir que aplicações consumidoras distintas não consomem o mesmo evento da fila?

A forma de funcionamento mais tradicional de um *broker* é excluir o evento após o consumo. Uma vez que uma das aplicações consumidora confirmou com sucesso o processamento ou pelo menos o recebimento, o evento é excluído e outros consumidores não conseguem processá-lo novamente.

Mas nem todos os *brokers* funcionam dessa forma. Alguns *brokers* não excluem o evento após a confirmação da aplicação consumidora. Eles definem um período máximo de retenção, o qual pode ser reconfigurado pelo cliente, respeitando o período máximo. Por exemplo, cada evento criado no Amazon SQS tem um limite máximo de 14 dias que o cliente pode configurar. O valor *default* são 4 dias. Os eventos no Amazon SQS são excluídos apenas quando acaba o período da retenção ou quando o cliente explicitamente remove o evento.

Para evitar que outros consumidores registrados na mesma fila processem o mesmo evento, o *broker* define um tempo *limite de visibilidade do evento*, um período durante o qual o *broker* impede que outros consumidores recebam e processem o evento. Quando a aplicação consumidora termina o processamento do evento dentro do período de visibilidade, ela remove o evento explicitamente via código para garantir que outros consumidores não peguem o evento para processar.

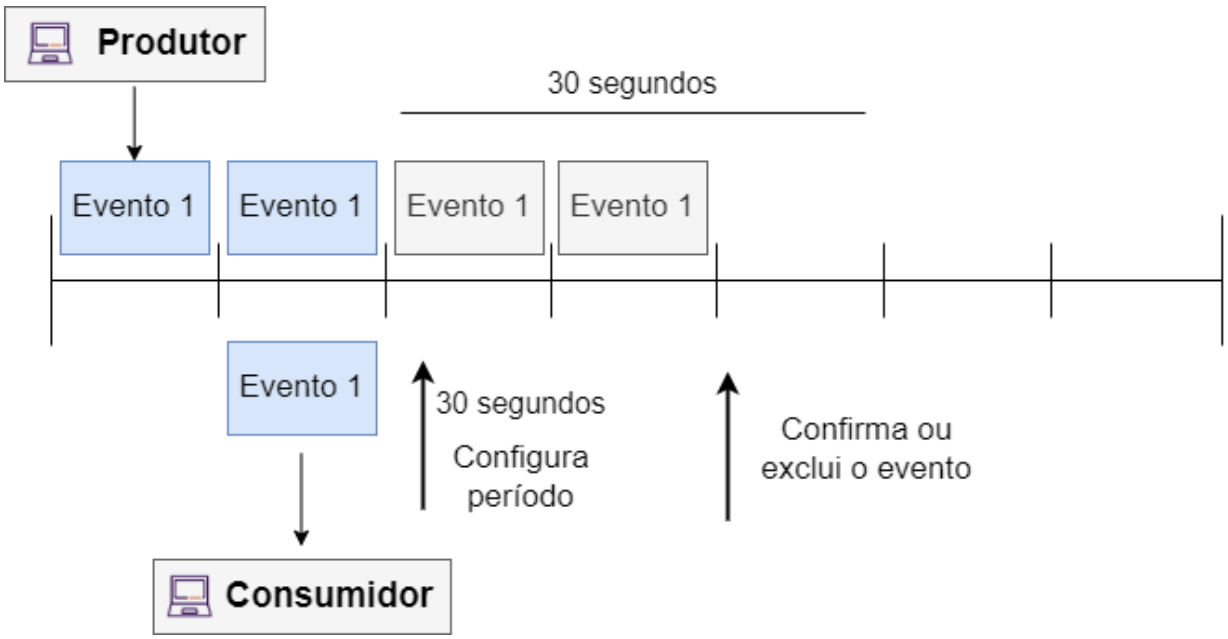

Figura 6.13: Visibilidade do evento – confirma ou exclui o evento dentro do período.

> Em alguns *brokers*, você não precisa explicitamente remover os eventos, apenas confirmá-los. Nesses casos, o *broker* marca o evento como entregue ou consumido ou confirmado e deixa o evento invisível para outras aplicações. Esse comportamento é para permitir a reprodução dos eventos de acordo com o período de retenção.

Em um cenário de falha da aplicação consumidora que está processando o evento, após o período de visibilidade do evento, o evento fica disponível para outros consumidores registrados da fila.

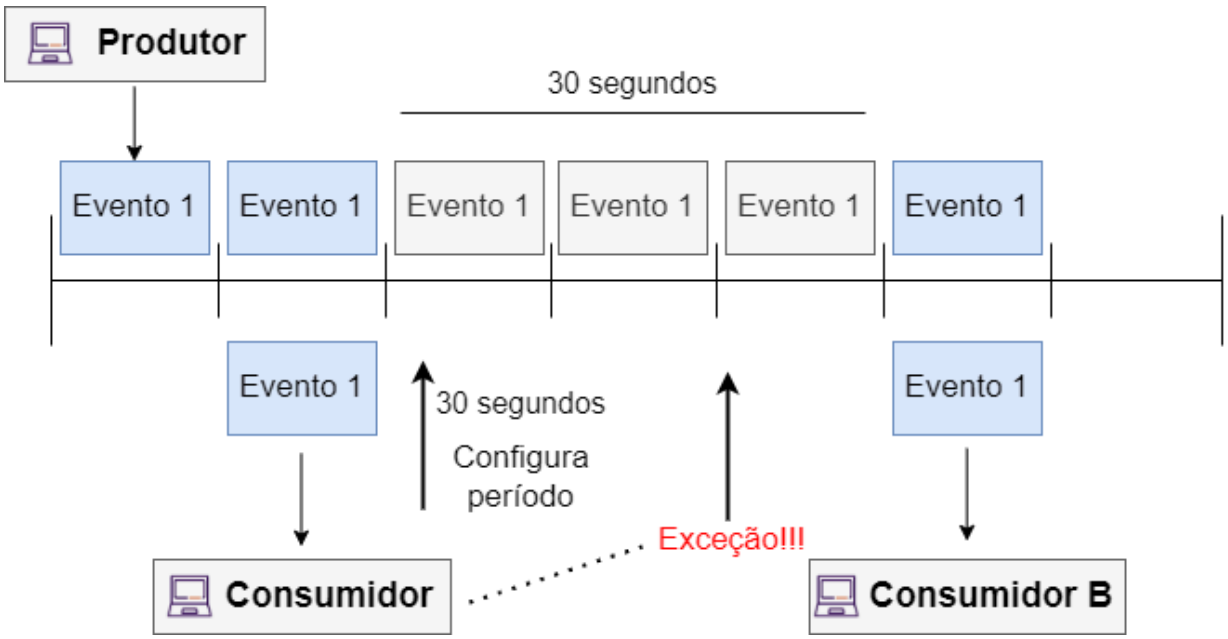

Figura 6.14: Visibilidade do evento – evento fica disponível após período por causa da não exclusão ou confirmação.

Determinar o valor dessa configuração é difícil. Se colocarmos um valor baixo, pode ocorrer de o evento ficar disponível para outros consumidores enquanto o primeiro ainda está processando. Nesse caso, teremos um cenário clássico de evento duplicado. Se colocarmos um valor muito alto, podemos atrasar o processamento do evento em cenário de falhas. Se a aplicação que estava em processamento falhar, como configuramos um valor alto, o evento vai demorar a ser processado por outra aplicação consumidora.

Para mitigar a dificuldade dessa configuração, alguns *brokers* permitem que estendamos o período de visibilidade do evento em tempo de execução. Isso torna o código mais complexo, mas dá à aplicação consumidora mais tempo para terminar seu processamento.

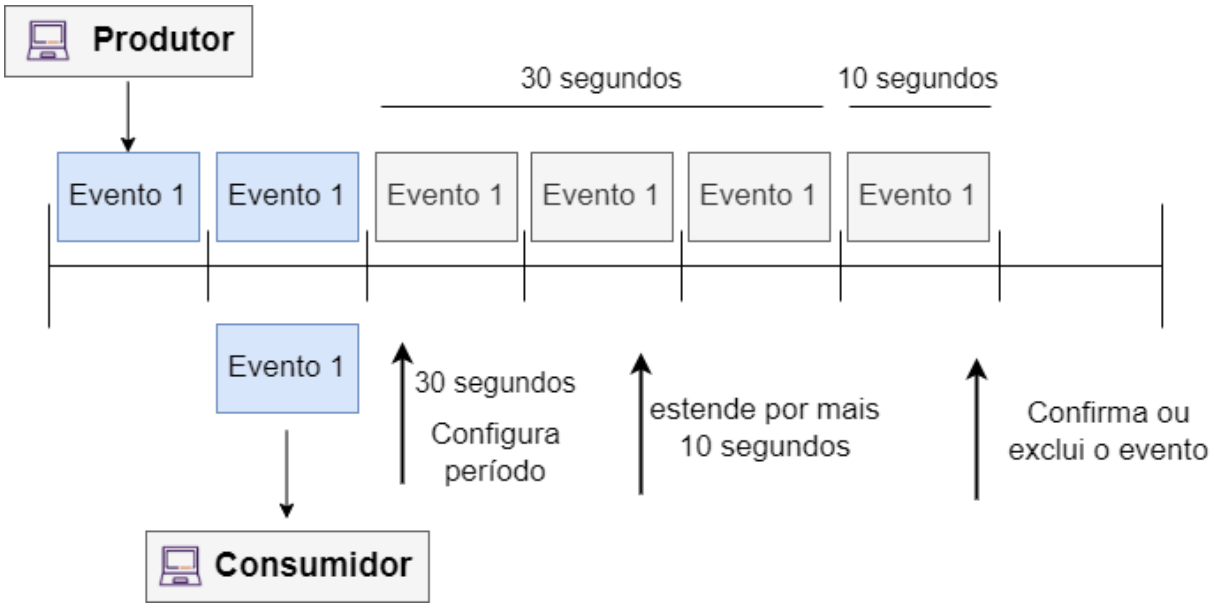

Figura 6.15: Visibilidade do evento – estender a visibilidade do evento.

6.9 Agendamento da entrega do evento

Por padrão, uma vez que o *broker* armazenou e confirmou o recebimento do evento para a aplicação produtora, ele imediatamente disponibiliza-o para a aplicação consumidora registrada processar. Mas às vezes, precisamos que os eventos produzidos não sejam imediatamente consumidos e processados pelas aplicações consumidoras. O evento pode ser *agendado* (*scheduled events*). Essa capacidade também é conhecida como *atraso na entrega do evento* (*delayed events*).

Imagine uma situação em que uma determinada aplicação produtora precise publicar uma grande quantidade de eventos em um curtíssimo intervalo de tempo. O *broker* utiliza o mecanismo de *push* para enviar esses eventos para um *endpoint* HTTPS da aplicação consumidora registrada.

No nosso caso, a aplicação consumidora é lenta, pois é uma API legada que não escala na mesma proporção e não consegue

executar todos os eventos de uma vez. Nessa situação, precisamos que o *broker* cadencie o envio desses eventos em um ritmo no qual a aplicação consumidora não fique sobrecarregada.

Essa configuração pode ser definida em nível de fila (que servirá para todos os eventos) ou a aplicação produtora pode fazer a definição para cada evento no momento da publicação. O agendamento do evento segue o mesmo princípio do *período de visibilidade do evento*, que vimos na seção anterior. Ele faz com que o evento fique invisível na fila para todos os consumidores registrados até o tempo configurado expirar.

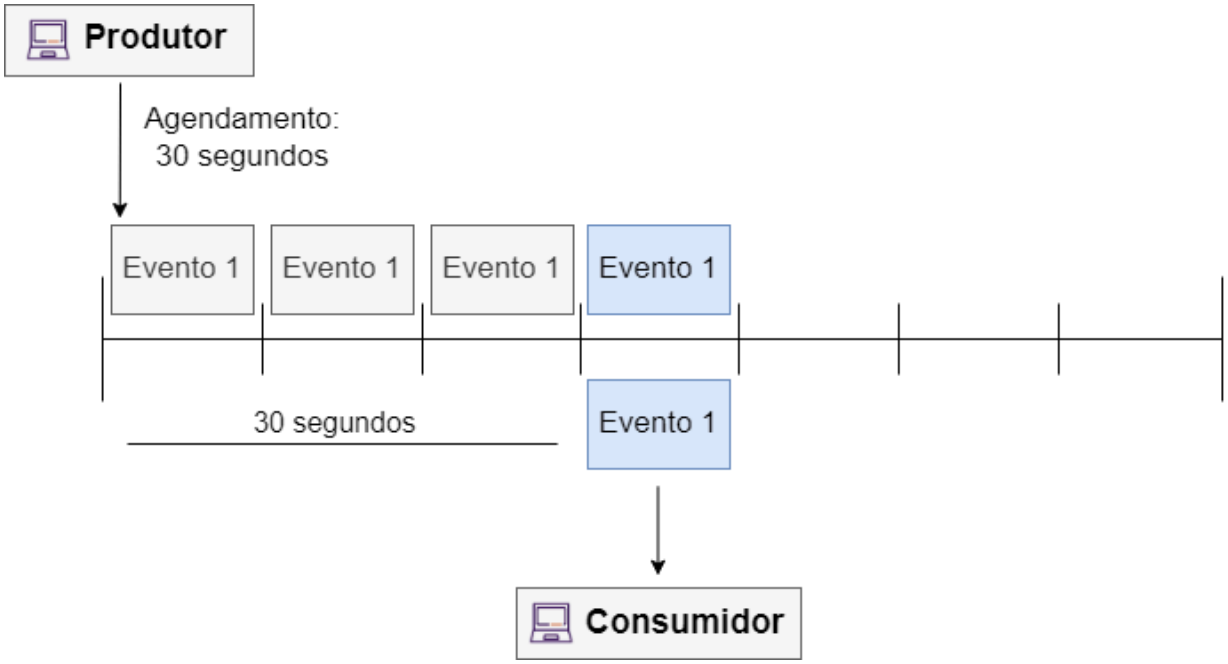

Figura 6.16: Atraso ou agendamento do evento.

Falamos de um exemplo com o mecanismo *push* para controlar o envio de eventos para uma API legada. Mas podemos ter situações em que não precisamos executar o evento em tempo real (assim que é recebido pelo *broker*). Podemos agendar os eventos com alguns segundos, minutos ou horas de acordo com a necessidade do negócio.

Soluções que precisam de processamento em tempo real tendem a ter um custo maior. Por exemplo, se nesse nosso caso precisássemos executar os eventos em tempo real e tivéssemos uma API escalável, possivelmente adicionaríamos mais instâncias da aplicação consumidora no mesmo grupo, seguindo o padrão de entrega *point-to-point*, para balancear a carga. Isso elevaria o custo da nossa solução, mas atenderia o negócio com processamento em tempo real dos eventos.

Concluindo

Neste capítulo, exploramos algumas das principais capacidades dos *brokers* de eventos relacionadas ao fluxo de publicação e consumo dos eventos. Vimos como funciona os mecanismos de entrega dos eventos *push* e *pull* e os casos de uso que são mais apropriados para cada um deles. Vimos também os SDKs, que usamos em nossas aplicações ou microsserviços para abstrair a complexidade do protocolo de comunicação.

Exploramos funcionalidades para minimizar o tráfego de rede na comunicação entre as aplicações e o *broker* e remover o desperdício de processamento para descarte de eventos, com o envio e recebimento de eventos em lote e roteamento inteligente, respectivamente. Em seguida, vimos como os modos de confirmação de recebimento dos eventos nos ajuda a determinar o nível da garantia de entrega para nossa solução. Vimos também a possibilidade de reter os eventos por tempo indeterminado no *broker* e a possibilidade de reproduzirmos todo o histórico dos eventos.

Por fim, vimos como evitar que outros consumidores registrados na mesma fila processem o mesmo evento com a configuração do limite da visibilidade do evento e como podemos agendar os eventos, em situações que não precisamos executá-los em tempo real.

No próximo capítulo, vamos entender os tipos de *brokers* de eventos existentes no mercado e explorar como as capacidades que vimos

até aqui se relacionam com cada tipo.

CAPÍTULO 7
Os tipos de brokers de eventos

Neste último capítulo desta parte sobre *broker* de eventos, vamos explorar os tipos de *broker* mais comuns e como os principais *brokers* de mercado lidam com eles.

> Vamos utilizar como referência os tipos de *broker* propostos pela Gartner em sua pesquisa mais recente, *Choosing Event Brokers: The Foundation of Your Event-Driven Architecture* (OLLIFFE, 2020).

Até agora, vimos diversas funcionalidades ou capacidades que um *broker* de eventos pode nos oferecer para implementarmos nossa solução de EDA. Essas funcionalidades estão relacionadas às características do tipo de *broker*.

Segundo a Gartner, os três tipos de *broker* de eventos mais comuns no mercado são **orientado a fila** (*queue-oriented*), **orientado a log** (*log-oriented*) e **orientado a assinatura** (*subscription-oriented*). Na verdade, o nome dos tipos é o que menos importa. O importante é a característica de cada um deles. São essas características que nos direcionam no processo de escolha do *broker*:

- *Orientado a fila*: Esses *brokers* focam na comunicação desacoplada e assíncrona entre as aplicações através de fila. Os eventos publicados são armazenados e **tipicamente removidos depois de confirmados pelas aplicações consumidoras**;
- *Orientado a log*: Esses *brokers* focam na **ingestão de eventos de série (*stream*) de forma contínua (*append-only*) e**

não removem os eventos depois de confirmados pelas aplicações consumidoras;
- *Orientado a assinatura*: Esses *brokers* focam na distribuição de eventos **baseados em regras de filtro** utilizando o **mecanismo de entrega *push***.

7.1 Broker orientado a fila

A característica que mais se destaca nesse tipo de *broker* é que ele tipicamente remove os eventos após a confirmação das aplicações consumidoras e não tem um período de retenção de eventos com longa duração. Esse tipo de *broker* é mais apropriado quando precisamos utilizar os padrões de entrega de eventos *point-to-point* e *request/reply*, por se tratar de um canal do tipo fila, como vimos na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*.

Mas esse tipo de *broker* também é apropriado para o padrão *publish/subscribe*. Uma característica comum desse tipo de *broker* para contemplar o padrão *publish/subscribe* é criar uma fila para cada aplicação consumidora (ou para o grupo de consumidores), juntamente a um mecanismo de roteamento de eventos para as filas apropriadas. Esta é uma característica típica do protocolo de mensagem AMQP, como vimos na seção *Protocolo de mensagem* do capítulo 2, *Entendendo os principais conceitos*.

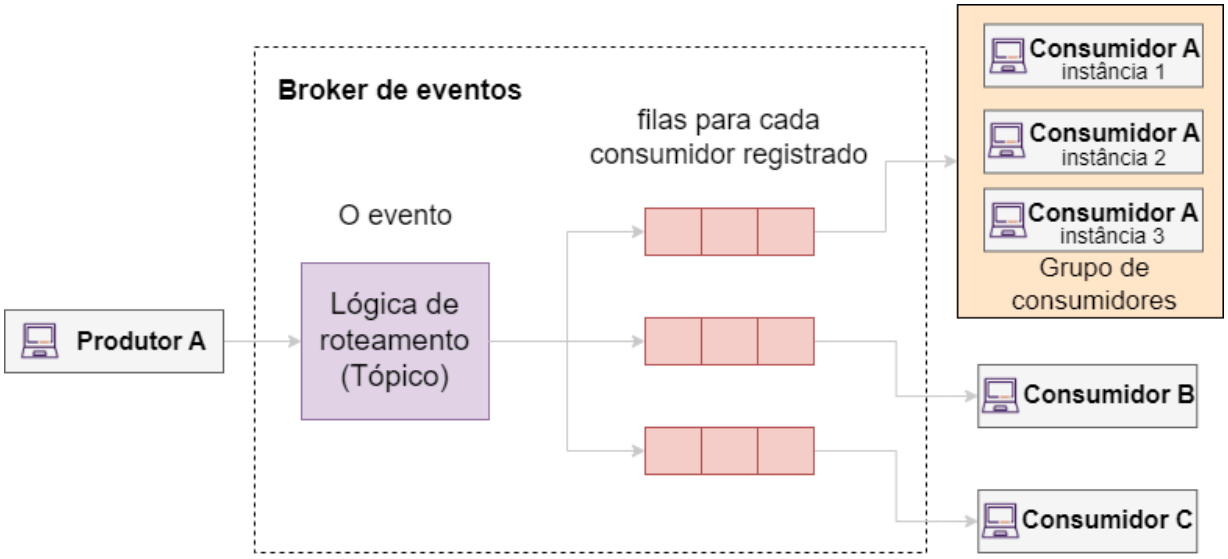

Figura 7.1: Broker orientado a fila.

Na perspectiva da aplicação produtora, o evento é publicado em um tópico, que encaminha o evento para a(s) fila(s) apropriada(s). Já na perspectiva da aplicação consumidora, elas estabelecem a conexão com o *broker*, que empurra o evento para elas utilizando o mecanismo de *push* (normalmente ocultado pelo SDK). As aplicações processam os eventos e enviam a confirmação para o *broker*. O *broker* então remove o evento da fila da aplicação consumidora.

A lista a seguir mostra os padrões e funcionalidades que são mais predominantes nos *brokers* orientados a fila. Isso não significa que *brokers* de outro tipos não tenham algumas das funcionalidades em comum:

- Preservar a ordem dos eventos (FIFO);
- *Dead-letter queues* (DLQ);
- *Point-to-point*;
- *Publish/subscribe*;
- *Request/reply*;
- Agendamento da entrega do evento;
- Visibilidade do evento;
- Roteamento inteligente.

Segundo a *Gartner*, um *broker orientado a fila* é mais indicado se o cliente precisa das seguintes combinações:

1. Suporte a uma variedade de protocolos de mensagens (*multiprotocolo*);
2. Estruturas de tópicos hierárquicos com roteamento inteligente;
3. Suporte ao padrão de entrega *publish/subscribe* em conjunto com a funcionalidade *First in, first out* (FIFO).

7.2 Broker orientado a log

A característica que mais se destaca nesse tipo de *broker* é que ele não remove os eventos após a confirmação das aplicações consumidoras, mas os mantêm no tópico por tempo indeterminado para serem reproduzidos a partir de qualquer posição. Esse tipo de *broker* é mais apropriado quando precisamos utilizar os padrões de entrega de eventos *publish/subscribe* e *event streaming*, como vimos na seção *Padrões de entrega de evento* do capítulo *3. Padrões de arquitetura para a arquitetura orientada a eventos*.

Os tópicos nesse tipo de *broker* são logs de eventos de série. Os eventos são continuamente anexados (*append-only*) no log. Isso significa que, quando você publica um novo evento no log, ele sempre vai para o final do log. As aplicações consumidoras podem consumir do log a partir de qualquer *offset*, que é um número incremental e imutável que corresponde à posição de cada registro/evento dentro do log.

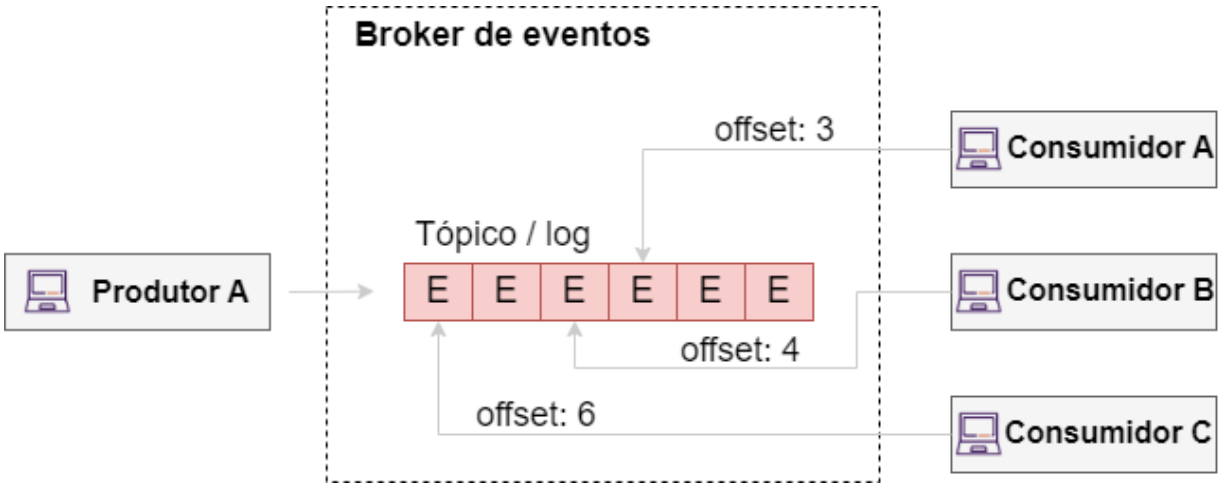

Figura 7.2: Broker orientado a log.

O tópico é um conceito lógico dentro desse tipo de *broker*. Os tópicos são divididos em *partições*, que são a quebra do log em pequenos logs, que efetivamente armazenam os dados. Os dados são replicados, através das partições, para vários servidores/nós dentro do *cluster*. A replicação dos dados através das partições é que faz esse tipo de *broker* ser distribuído, altamente escalável e tolerante a falhas.

Na perspectiva das aplicações produtoras, elas sabem apenas sobre tópicos. Por trás dos panos, o evento é redirecionado para uma das partições utilizando o método *round-robin*. Os eventos nesse tipo de *broker* são enviados como par de chave/valor (*key/value*).

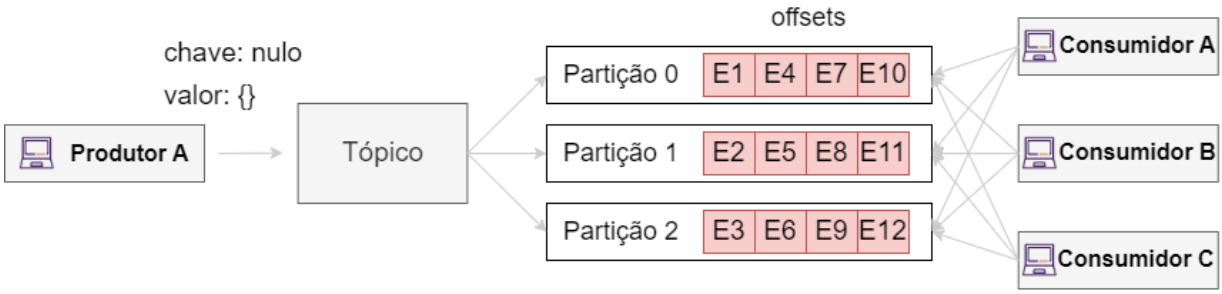

Figura 7.3: Broker orientado a log – partições.

Através das partições, conseguimos preservar a ordem dos eventos. Para garantir que o mesmo conjunto de eventos (ex.: os pedidos do

mesmo cliente) seja redirecionado para a mesma partição no momento da publicação, a aplicação deve enviar a mesma chave (ex.: o cliente ID com valor `1234`):

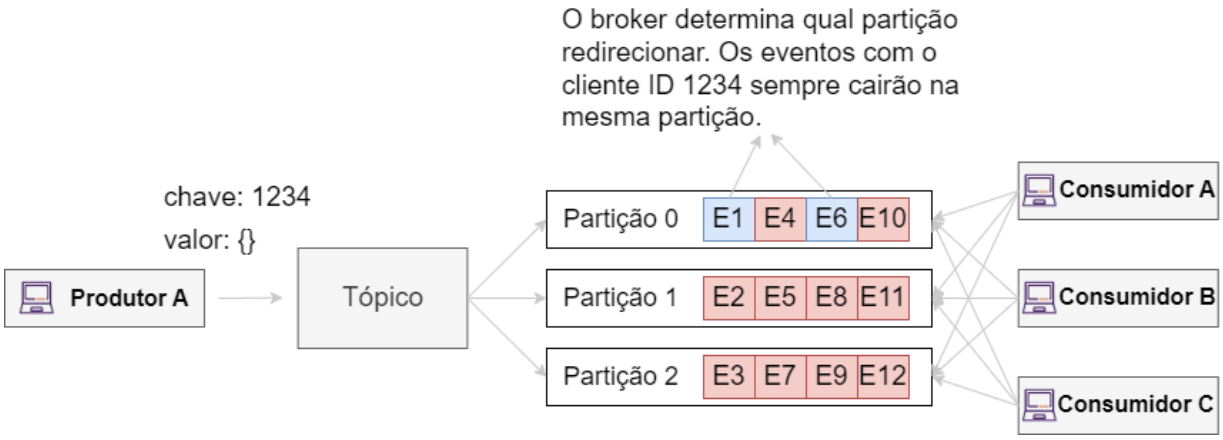

Figura 7.4: Broker orientado a log – ordem dos eventos preservada por partição

Já na perspectiva das aplicações consumidoras, elas estabelecem a conexão com o *broker* e consomem o evento (individual ou em lote) a partir do *offset*, utilizando o mecanismo de *pull*. Elas processam o evento e confirmam o consumo (ou *commit log*) do *offset* e vão para o próximo offset/evento.

As aplicações consumidoras podem iniciar o consumo dos eventos a partir de qualquer *offset*, que corresponde à posição do log, como vimos na seção *Reprodução do evento* do capítulo 6, *Funcionalidades comuns de um broker no fluxo de publicação e consumo de eventos*. Quando uma aplicação consumidora consome um tópico, ela consome de todas suas partições.

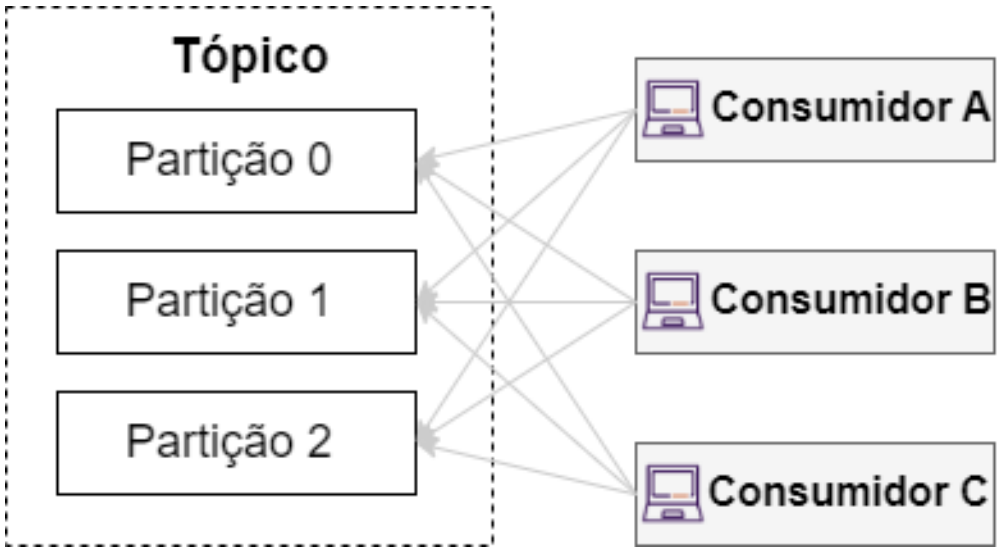

Figura 7.5: Broker orientado a log – perspectiva da aplicação consumidora.

Quando uma aplicação consumidora tem diversas instâncias no mesmo grupo de consumidores para paralelizar o processamento dos eventos ou até mesmo diferentes aplicações no mesmo grupo, o *broker* faz a proporção da forma mais justa possível para que cada aplicação fique alocada a uma e somente uma partição.

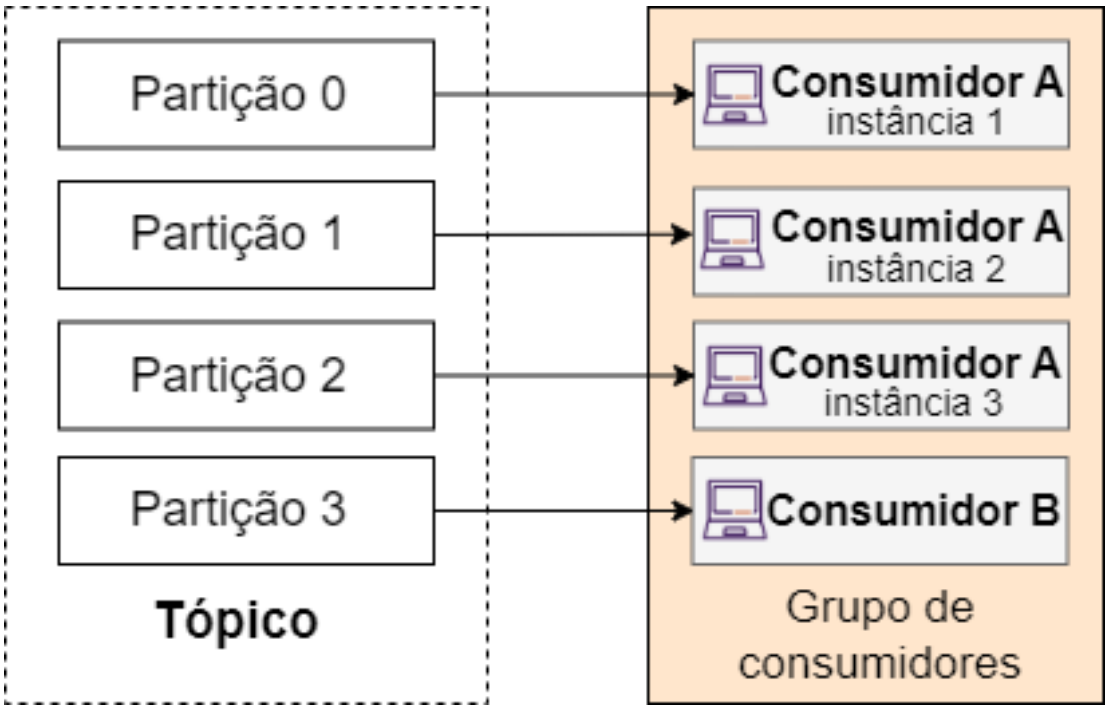

Figura 7.6: Broker orientado a log – alocação de partição com aplicações do mesmo grupo.

Se tivermos mais aplicações do que partições, a aplicação ficará inativa (*idle*).

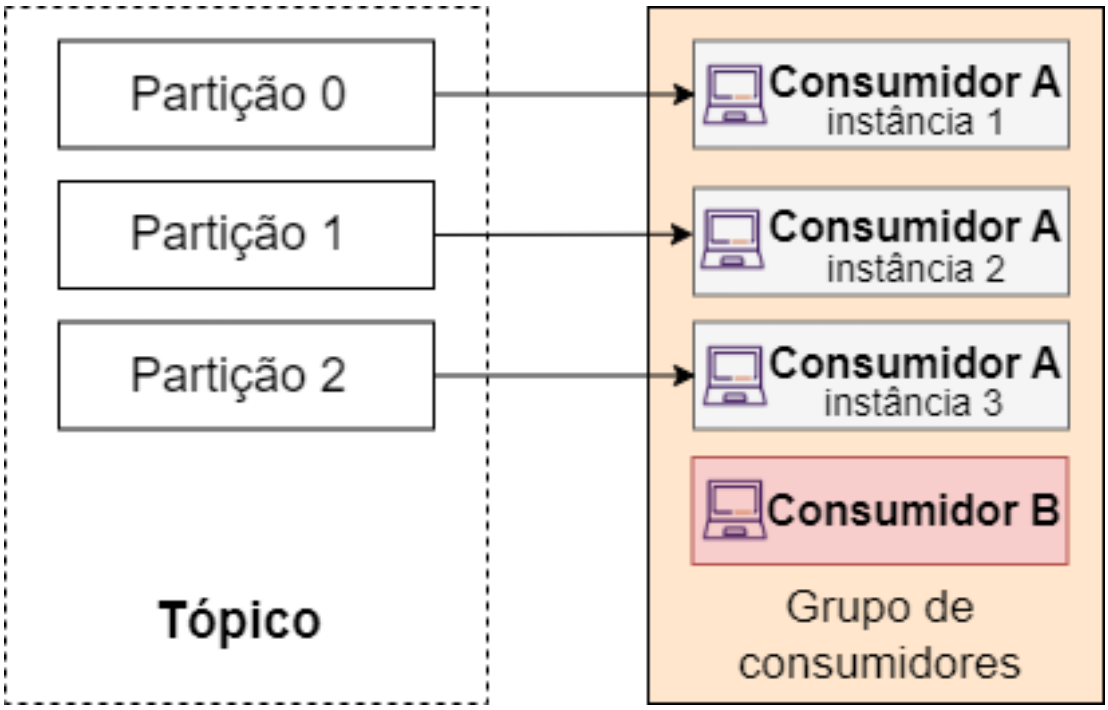

Figura 7.7: Broker orientado a log – alocação de partição com uma aplicação inativa.

Caso tenhamos mais partições do que aplicações, as partições serão alocadas às aplicações da forma mais justa possível (neste exemplo, cada aplicação está alocada em duas partições):

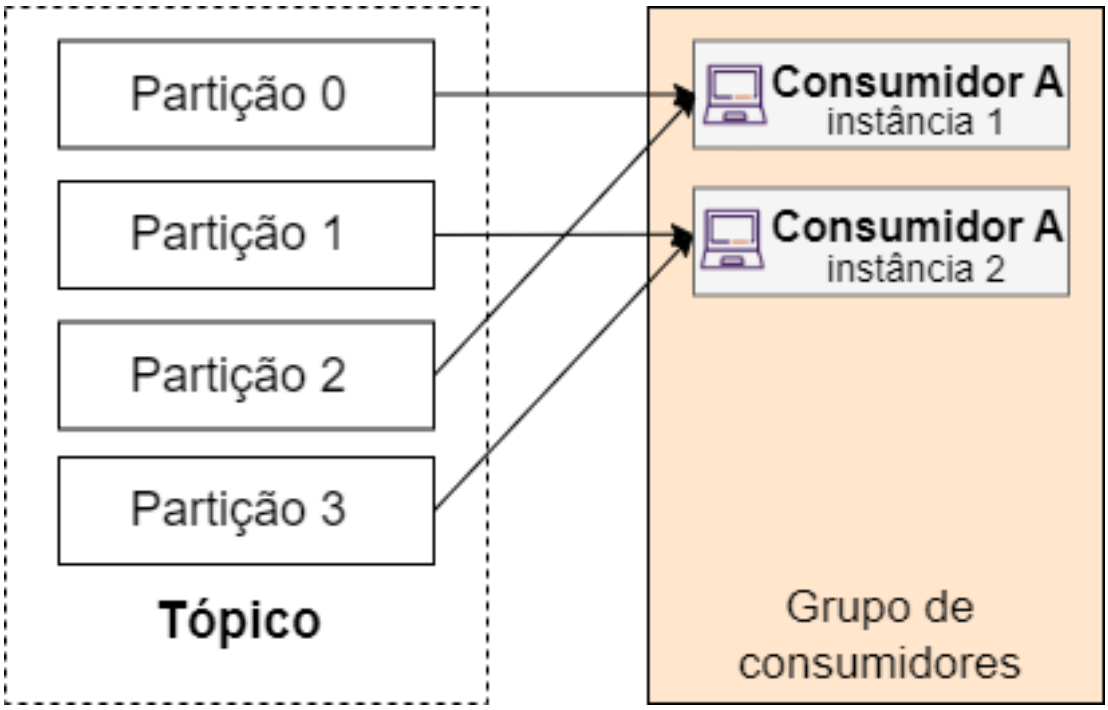

Figura 7.8: Broker orientado a log – mais partição do que aplicação para alocação.

Quando adicionamos novas aplicações no grupo, ou quando uma aplicação deixa de fazer parte do grupo, o *broker* faz o reequilíbrio (*rebalance*) das alocações para manter a paridade entre as partições e as aplicações.

Agora que sabemos as características desse tipo de *broker*, a lista a seguir mostra os padrões e funcionalidades que são mais predominantes:

- *Event streaming*;
- *Publish/subscribe*;
- Retenção do evento;
- Reprodução do evento;
- Gerenciamento de esquema;
- Envio e recebimento de eventos em lote;
- *Event Sourcing*;
- Preservar a ordem dos eventos (através da partição).

Segundo a Gartner, um *broker orientado a log* é mais indicado quando precisamos:

1. Reter os eventos por uma longa duração ou por tempo indeterminado;
2. Reproduzir os eventos a partir de qualquer posição do log (*offset*);
3. Atender a um volume muito grande de eventos de série que promovam o paralelismo do consumo dos eventos através das partições.

7.3 Broker orientado a assinatura

A característica que mais se destaca nesse tipo de *broker* é que ele foca na distribuição de eventos com base em regras de filtro utilizando o mecanismo de entrega *push*. São mais apropriados para quando precisamos utilizar os padrões de entrega de eventos *publish/subscribe*, como vimos na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*.

Cada tópico tem uma assinatura (*subscription*) associada. O cliente define uma regra de filtro (entre outras funcionalidades) que o *broker* utilizará para encaminhar para a aplicação consumidora registrada nessa assinatura.

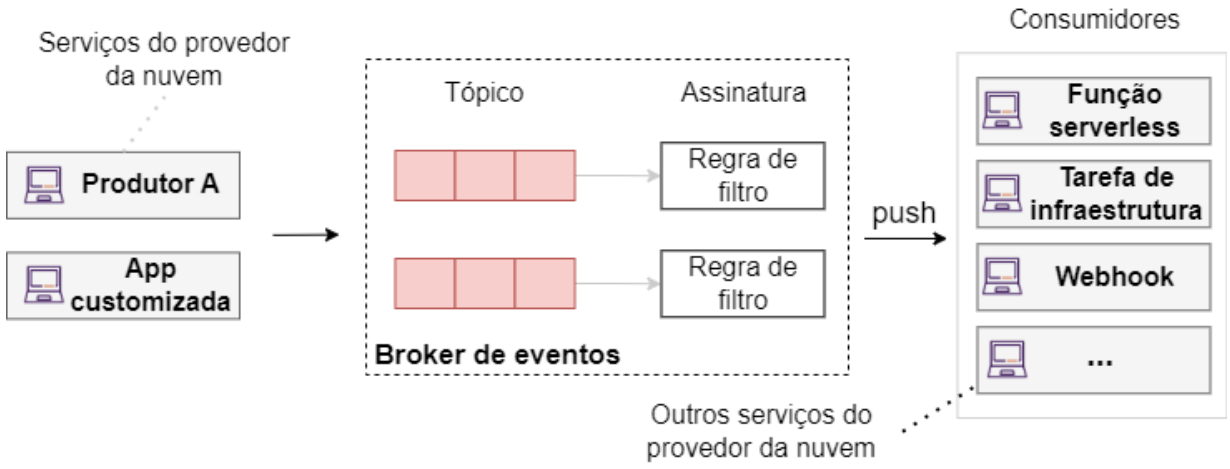

Figura 7.9: Broker orientado a assinatura.

Na perspectiva da aplicação produtora, ela pode ser um serviço do provedor da nuvem, que tem uma integração nativa com o *broker*, ou uma aplicação customizada, que utiliza o SDK do *broker* para publicar o evento. Alguns fornecedores podem prover funcionalidades de retentativa caso não recebam a confirmação de recebimento do *broker*. Já na perspectiva da aplicação consumidora, o *broker* conecta nas aplicações registradas para entregar os eventos com o mecanismo *push*. As aplicações confirmam o processamento do evento para o *broker*. Essas aplicações também podem ser serviços do provedor da nuvem ou *webhooks*.

Uma vez que o cliente esteja utilizando serviços específicos do provedor da nuvem, uma grande vantagem em usar esse tipo de *broker* é sua integração nativa com outros serviços do provedor. Por exemplo, se a solução de EDA do cliente está baseada nos serviços da AWS, como *AWS Lambda*, *Step Functions*, *Fila do SQS*, *Tópico do SNS*, *API Gateway*, entre outros, uma boa opção é utilizar o *broker orientado a assinatura* da AWS.

Agora que sabemos as características desse tipo de *broker*, a lista a seguir mostra os padrões e funcionalidades que são mais predominantes:

- *Publish/subscribe*;

- *Webhooks*;
- Roteamento inteligente;
- Mecanismo *push*;
- Agendamento da entrega do evento;
- Retentativa.

Segundo a Gartner, um *broker orientado a assinatura* é uma boa opção quando:

1. O cliente está construindo uma solução EDA nativa da nuvem em um único ambiente de provedor de nuvem (por exemplo, AWS);
2. A solução de EDA exigir apenas a semântica de *entrega pelo menos uma vez*;
3. A funcionalidade de ordenação dos eventos (FIFO) não for um requisito para o negócio;
4. O cliente implementar automação de infraestrutura no ambiente do provedor da nuvem;
5. Implementar uma arquitetura *serverless* que depende (pelo menos parcialmente) dos eventos publicados por serviços do provedor da nuvem. Por exemplo, publicar um evento no *broker orientado a assinatura* da AWS quando um arquivo for adicionado em um determinado *bucket* do Amazon S3.

7.4 Como os fornecedores lidam com os diferentes tipos de brokers

O objetivo desta última seção é associar os principais *brokers* de eventos do mercado aos tipos de *broker* que vimos nas seções anteriores. Essa associação nos dará uma visão ampla das características mais predominantes de cada *broker* do mercado e auxiliará o processo de escolha para nossa solução de EDA:

- **Orientado a fila**: Amazon SQS, Amazon MQ, Apache ActiveMQ, Google Cloud Pub/Sub, IBM MQ, Microsoft Azure Service Bus, RabbitMQ e Solace PubSub+;
- **Orientado a log**: Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka (MSK), Apache Kafka, Confluent e Microsoft Azure Event Hubs;
- **Orientado a assinatura**: Amazon SNS, Amazon EventBridge, Google Cloud Eventarc, Microsoft Azure Event Grid e Sensedia Events Hub.

Em um mercado cada vez mais competitivo e diversificado, os fornecedores de *brokers* tentam atender a uma maior variedade de casos de uso possível. Fazer uma categorização perfeita é quase impossível. Alguns *brokers* têm funcionalidades que abrangem mais de um tipo. Por exemplo:

- O RabbitMQ tem o tipo predominante orientado a filas, mas oferece plug-in para suportar eventos de série (*stream*);
- O Amazon EventBridge é predominantemente orientado a assinatura, mas tem funcionalidades de gerenciamento de esquema, retenção e reprodução de eventos;
- O Google Cloud Pub/Sub tem o tipo predominante orientado a fila, mas tem algumas funcionalidades típicas de um *broker* orientado a log e assinatura, como reter e reproduzir eventos até um certo período; e fazer notificação utilizando o mecanismo de *push* a um *endpoint* HTTPS;
- O Solace PubSub+ tem o tipo predominante orientado a fila, mas permite reter e reproduzir eventos.

O contrário também é verdade. Alguns *brokers* que têm características predominantes de um determinado tipo podem não ter algumas funcionalidades características do tipo em questão. Por exemplo:

- O Amazon SQS não remove o evento automaticamente após o consumo. O evento é removido após o período de retenção ou

quando a aplicação consumidora explicitamente o remove via código;
- O Sensedia Events Hub, que é o único *broker* brasileiro da lista, apesar de estar associado ao tipo orientado a assinatura, não tem a funcionalidade de regras de filtro.

Com a categorização dos *brokers* de mercado com os tipos de *brokers* que vimos nas seções anteriores, podemos tirar algumas conclusões, como:

1. O principal propósito do RabbitMQ, que é orientado a fila, não é armazenar eventos por um longo período para serem reproduzidos, mas entregar eventos para as aplicações consumidoras registradas e em seguida removê-los;
2. O caso de uso ideal para o Apache Kafka, que é orientado a log, é processar fluxo contínuo de eventos de série (*streaming* de eventos) com possibilidade de reproduzi-los a partir de qualquer posição do log;
3. Apesar de um *broker* de eventos ter sua característica mais predominante para o tipo orientado a log, ele consegue atender casos de uso orientado a fila;
4. Se minha solução é baseada na arquitetura *serverless* utilizando o provedor de nuvem AWS, posso utilizar o *broker* Amazon EventBridge, que é orientado a assinatura, para ter uma integração nativa com outros serviços da AWS da minha solução.

Concluindo

A visão ampla das características dos principais *brokers* de mercado, de acordo com os tipos de *brokers* que vimos neste capítulo, nos dá a capacidade de estabelecer um plano inicial de quais *brokers* de eventos provavelmente atenderão às necessidades do cliente.

Essa pode ser considerada a etapa inicial do processo de escolha do *broker* para determinar as diretrizes. É necessário analisarmos cada funcionalidade e capacidade que vimos no decorrer do livro, como

padrões de entrega de eventos, padrões de implantação (*deploy*), operação, governança, gerenciamento de esquema, funcionalidades referentes ao fluxo de publicação e consumo de eventos, entre outras, para determinar o *broker* de eventos mais apropriado para atender aos requisitos de negócio do cliente.

É comum que na mesma empresa tenhamos diversos projetos/soluções de EDA. Cada solução tem necessidades e características distintas e, com isso, podemos utilizar mais de um fornecedor de *broker* para atender às iniciativas de EDA da empresa.

Neste capítulo, categorizamos os principais *brokers* de eventos de mercado baseado nos tipos de *brokers* sugeridos pela Gartner – *orientado a fila*, *orientado a log* e *orientado a assinatura* – para entendermos as características predominantes de cada *broker*.

No próximo capítulo, começa a parte III do livro, onde vamos explorar outras arquiteturas e plataformas que se beneficiam da EDA.

Parte III: Arquitetura orientada a eventos e além

Capítulo 8
Padrões arquiteturais que se beneficiam da Arquitetura orientada a eventos

Bem-vindo(a) à parte III do livro!

Como falamos nos capítulos anteriores, EDA não é uma solução *stand-alone*. Isso significa que ela é implementada em conjunto com outros tipos de arquiteturas e plataformas.

As arquiteturas e plataformas que veremos nesta parte do livro (capítulos 8 e 9) não são sobre EDA, mas elas têm uma relação muito próxima com a EDA. Por esse motivo, aqui teremos uma visão geral delas, já que são consideradas as principais arquiteturas e plataformas utilizadas em diversos projetos de TI. Vamos mostrar como a EDA pode ser usada em conjunto e os benefícios que são gerados ou potencializados por ela.

O conhecimento adquirido nesta parte do livro nos ajudará na implementação dos projetos da parte IV. Nas referências bibliográficas, no final do livro, há sugestões de leitura para aprofundamento nessas arquiteturas e plataformas.

8.1 Arquitetura de microsserviços

A **arquitetura de microsserviços** é um estilo arquitetural no qual a solução é estruturada em inúmeras aplicações interagindo entre si através da rede para atingir um objetivo de negócio.

A quebra da solução em diversas aplicações independentes nos permite entregar de forma mais rápida e frequente soluções grandes e complexas. Essas aplicações independentes são pequenas aplicações (microsserviços) autônomas, implantadas e escaladas de forma independente e de fácil manutenção e teste.

A comunicação entre essas aplicações é tradicionalmente realizada de forma síncrona, usando o padrão mais tradicional de *request-response*. Esse tipo de comunicação faz com que as aplicações tenham um forte acoplamento, o qual dificulta a escalabilidade, a extensibilidade e a disponibilidade da solução.

A imagem a seguir é um exemplo de referência de uma arquitetura de microsserviços no padrão *request-response* sem EDA.

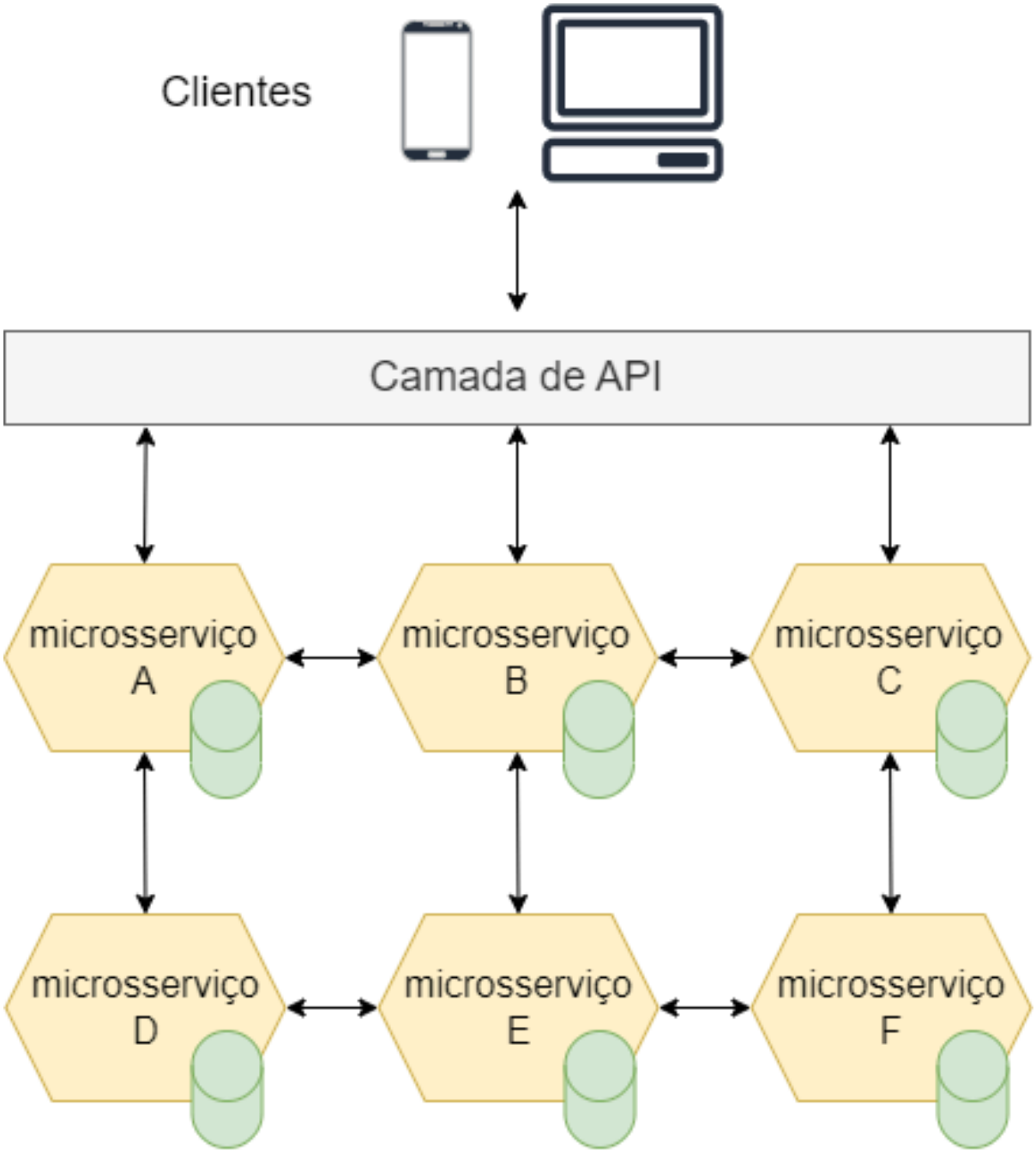

Figura 8.1: Exemplo de referência de uma arquitetura de microsserviços sem EDA.

A camada de API é a ponte (*middleware*) entre os clientes e os microsserviços. Essa camada tem várias responsabilidades importantes. Vamos explorar essas responsabilidades no próximo capítulo. Os microsserviços se comunicam entre si de forma síncrona com o padrão *request/response* para atingir um objetivo de negócio.

Esse tipo de comunicação faz com que o microsserviço obrigatoriamente esteja operante no momento em que é invocado. Dependendo da complexidade das invocações entre eles, se um dos microsserviços estiver com problema ou inoperante, isso vai desencadear erros generalizados, conhecidos como *falhas em cascata* (*cascading failures*), que têm impacto na saúde e na confiabilidade da solução como um todo.

A comunicação através de eventos ou mensagens não é um conceito novo, mas a necessidade de processarmos um grande volume de dados, em escala, com alta disponibilidade e em tempo real, é nova. A EDA potencializa os benefícios da arquitetura de microsserviços, como vimos na seção *Seus principais benefícios e desafios envolvidos* do capítulo 1, *Por que Arquitetura orientada a eventos?*. A imagem a seguir é um exemplo de referência de uma arquitetura de microsserviços com EDA.

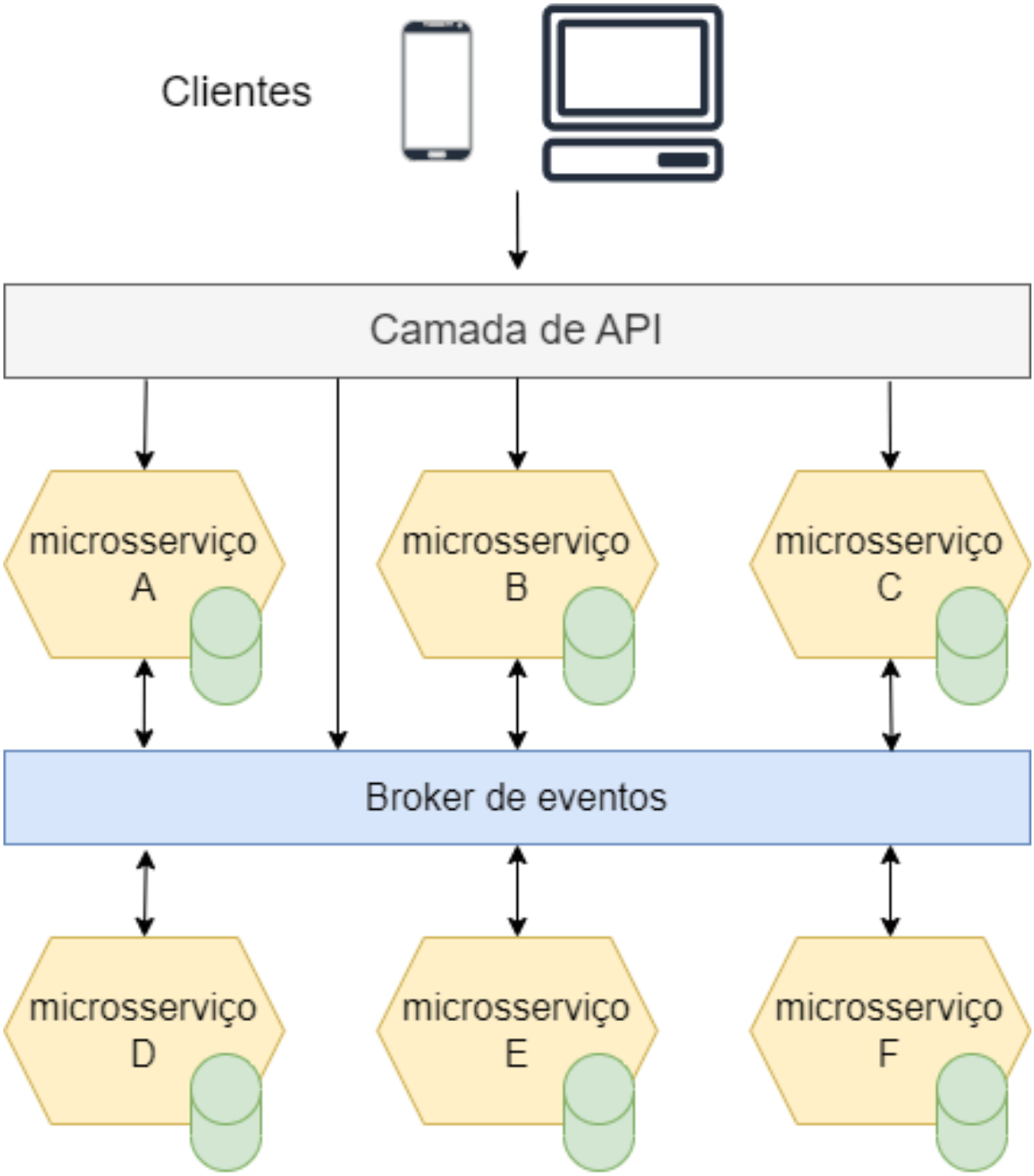

Figura 8.2: Exemplo de referência de uma arquitetura de microsserviços com EDA.

Os microsserviços se comunicam entre si de forma assíncrona com EDA através do *broker* de eventos para atingir um objetivo de negócio. A camada de API também pode publicar eventos para iniciar um fluxo.

A EDA traz uma maior confiabilidade da solução, pois se o microsserviço consumidor ficar inoperante, o *broker* retém os eventos, que ainda podem ser processados posteriormente quando o microsserviço consumidor voltar a ficar operante.

É muito provável que o cliente utilize os dois padrões de comunicação em sua solução com a arquitetura de microsserviços: a comunicação síncrona com *request-response* e a comunicação assíncrona com EDA. A recomendação é utilizar o padrão *request-response* na parte da solução onde os dados precisam ter uma consistência forte ou quando o usuário ou aplicação precisa aguardar por uma resposta imediata de uma solicitação (como uma *query*). Caso contrário, dê preferência à comunicação assíncrona com EDA, dados os benefícios potencializados por ela que vimos anteriormente.

8.2 Arquitetura serverless

A forma mais tradicional de executarmos o código/aplicação que desenvolvemos é nos servidores da empresa ou de algum provedor de nuvem. Precisamos definir o sistema operacional desejado, provisionar a capacidade de CPU, memória e instalar todos os softwares necessários para que o código seja executado.

O fato é que o provisionamento e manutenção desses servidores, bem como toda infraestrutura que os sustentam, são de responsabilidade do cliente (ou parcialmente do cliente, se o código estiver sendo executado no servidor virtual de algum provedor de nuvem), fazendo com que o cliente não foque apenas no que realmente importa: as regras de negócio.

A arquitetura *serverless* (ou arquitetura sem servidor) é quando o cliente não precisa se preocupar com o servidor onde seu código é executado. Ele utiliza soluções dos provedores de nuvem, como

AWS, Microsoft Azure e Google Cloud, que ficam com todo o trabalho de provisionamento, gerenciamento, manutenção e dimensionamento (*scale*) da infraestrutura para garantir que o código seja executado de acordo com a demanda do seu negócio. Isso faz com que o cliente foque apenas na construção do código e, consequentemente, nas regras do seu negócio.

A primeira coisa que vem à mente quando lemos ou ouvimos falar sobre *serverless* são as **funções como serviços** – ou, em inglês, *Functions-as-a-Service* (*FaaS*) – que os provedores de nuvem oferecem, como AWS Lambda (https://aws.amazon.com/pt/lambda/), Microsoft Azure Functions (https://azure.microsoft.com/pt-br/products/functions/) e Google Cloud Functions (https://cloud.google.com/functions?hl=pt-br).

Mas essas funções como serviço são apenas um dos tipos dentro de uma solução *serverless*. Além da computação sem servidores, através das funções, podemos ter armazenamento de dados sem servidor, *broker* de eventos sem servidor e por aí vai. Apenas para citar algumas, o Amazon EventBridge é uma solução de *broker* de eventos sem servidor, e o Amazon Aurora é uma solução de banco de dados sem servidor. Não importa o tipo de solução *serverless*, o princípio é o mesmo: o cliente não precisa se preocupar com a infraestrutura para executar o seu negócio.

A imagem a seguir é um exemplo de referência de uma arquitetura *serverless* utilizando uma camada de API, *broker* de eventos, banco de dados e funções, todos *serverless* utilizando um provedor de nuvem.

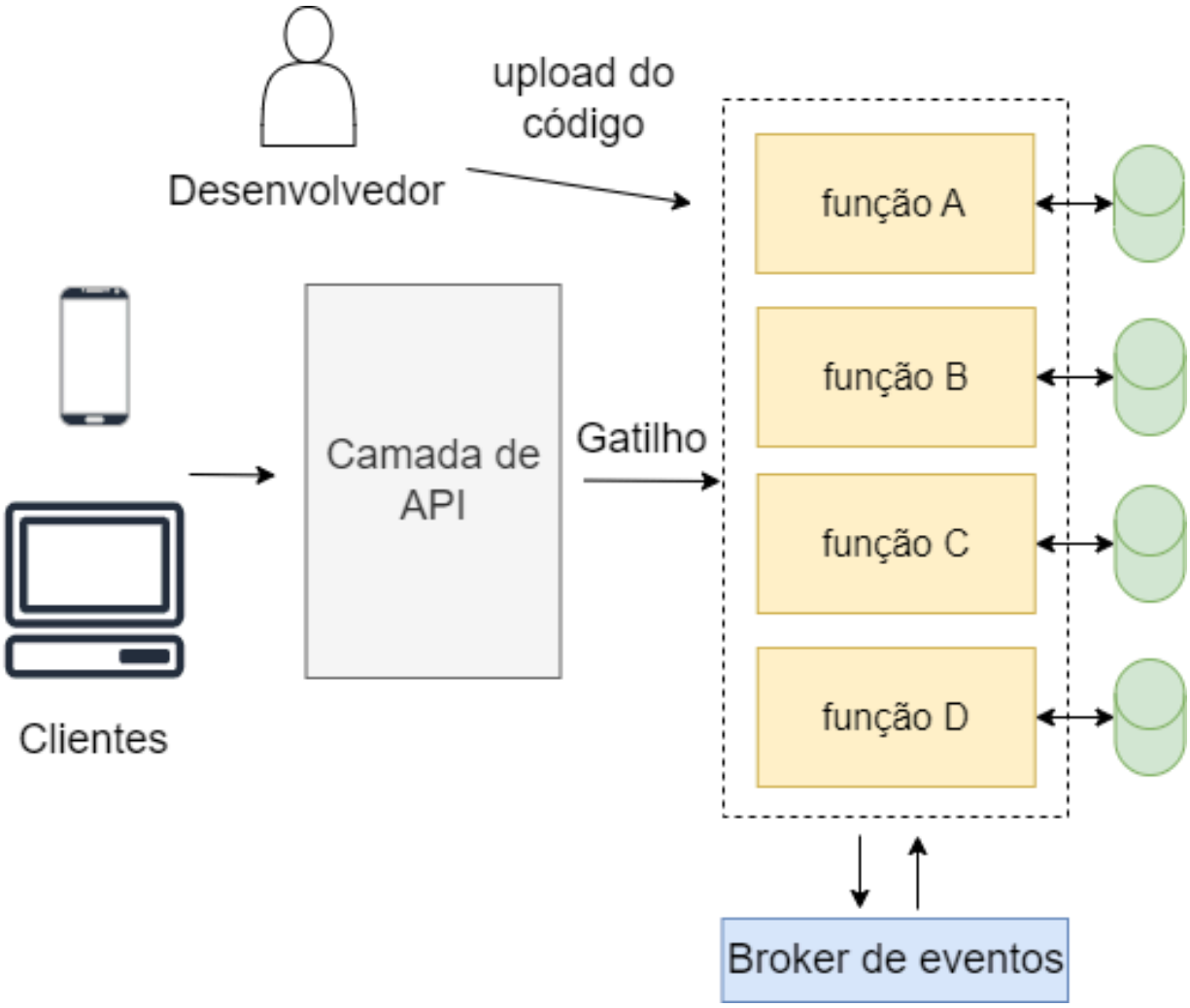

Figura 8.3: Exemplo de referência de uma arquitetura serverless.

A arquitetura *serverless* tem uma relação muito forte com a EDA. A comunicação através de eventos entre os componentes é tema recorrente quando o assunto é arquitetura *serverless*. O desenvolvimento de funções como serviço ou qualquer outro tipo de soluções sem servidores utilizando EDA é a base de uma arquitetura moderna nativa na nuvem. Tanto que, no lançamento do AWS Lambda no *AWS re:Invent* de 2014, a divulgação do produto, em tradução livre, foi: "*AWS Lambda – Um* ***serviço de computação orientado a eventos*** *para aplicações dinâmicas*" (https://www.youtube.com/watch?v=9eHoyUVo-yg).

A Gartner afirma, em sua pesquisa *Predicts 2022: Modernizing Software Development is Key to Digital Transformation* (2021a), que, até 2025, 60% das novas soluções de EDA usarão arquitetura *serverless*, devido a sua rápida elasticidade, seu modelo de precificação e sua baixa sobrecarga operacional.

Os benefícios que a EDA traz para a arquitetura *serverless* seguem os mesmos princípios trazidos para a arquitetura de microsserviços, que vimos na seção anterior.

8.3 Arquitetura de streaming de dados

A importância dos dados gerados pelas organizações nunca ficou tão evidente nos últimos anos. Eles chegam em uma quantidade infinita de fontes: sensores de IoT, servidores, logs de aplicações internas e externas, clientes, fornecedores, parceiros etc.

Esses dados são como um tesouro e, normalmente, são pouco explorados pelas empresas. Se esses dados forem processados em tempo hábil, eles podem trazer diversos benefícios para as organizações, como ajudá-las a melhorar o atendimento ao cliente, detectar e prevenir fraudes, aumentar a retenção de clientes, entender o desempenho das suas soluções digitais, melhorar processos de marketing, vendas, negócios e por aí vai.

Tradicionalmente, esses dados são armazenados em arquivos e/ou banco de dados SQL (*data at rest*) no decorrer do dia e processados na madrugada por um sistema (*batch job*) de extração, transformação e carregamento (do inglês, *extract, transform, and load*, ou ETL). Como resultado, os dados extraídos, transformados e enriquecidos são disponibilizados para análise através de *dashboards*.

Mas, muitas vezes, as empresas simplesmente não podem esperar que esses dados sejam processados no dia seguinte (o famoso *D+1*)

ou mesmo com algumas horas de defasagem. Elas precisam que eles sejam disponibilizados em tempo real para tomada de decisão. **Uma arquitetura de *streaming* de dados veio exatamente para atender a essa necessidade, disponibilizando os dados em tempo real por meio da EDA.**

Esse tipo de arquitetura é implementado por plataformas de *streaming* de dados, também conhecidas como plataformas de *streaming* de eventos, pois os dados são transmitidos através de eventos de série (*stream*), como vimos na parte de *Event Streaming* na seção *Padrões de entrega de evento* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*.

Plataformas de *streaming* de eventos normalmente são divididas em camadas ou etapas. As principais são:

1. **Captura**: etapa responsável pela captura dos dados de uma variedade de fontes, termo também conhecido como ingestão de dados (*data ingestion*);
2. **Armazenamento**: etapa responsável pelo armazenamento dos dados ou dos eventos de série (*data storage* ou *stream storage*);
3. **Processamento**: etapa responsável pelo processamento (*data processing* ou *stream processing*) de um grande volume de dados em tempo real. O processamento transforma os dados em informações relevantes para o negócio. O processamento pode envolver vários tipos de operações, tais como: validação, limpeza, normalização, transformação e enriquecimento dos dados;
4. **Envio**: etapa responsável pelo envio dos dados processados para os sistemas destinos.

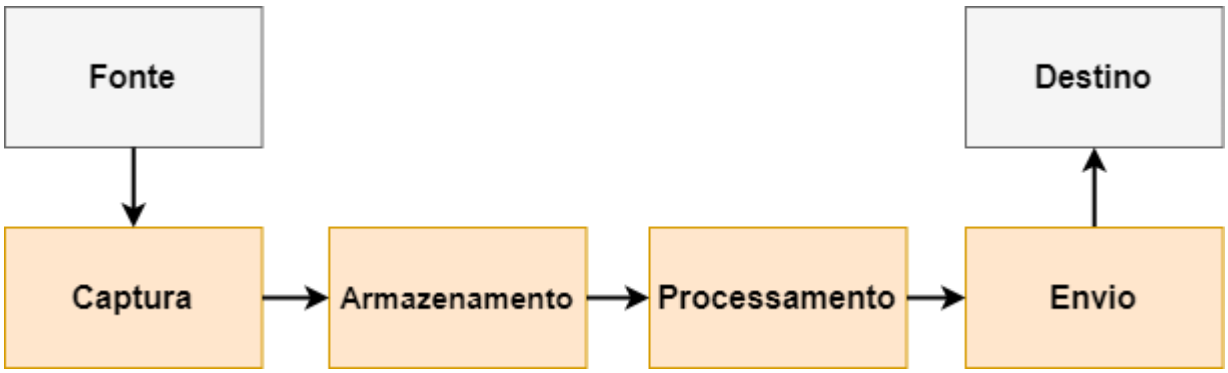

Figura 8.4: Etapas normalmente utilizadas por plataformas de streaming de eventos.

Projetos que utilizam esse tipo de plataforma para processar eventos de série em tempo real são conhecidos como projetos de *streaming* de dados (*data streaming*), projetos de *streaming* de eventos (*event streaming*), projetos de *pipelines* de dados (*data pipelines*) ou projetos de análise em tempo real (*real-time analytics*).

A imagem a seguir é um exemplo de referência de uma arquitetura de *streaming* de eventos utilizando a plataforma do Apache Kafka. Apesar de o *broker* de eventos ser o principal componente do Apache Kafka, ele tem outros componentes que contemplam as outras etapas de uma plataforma de *streaming* de eventos:

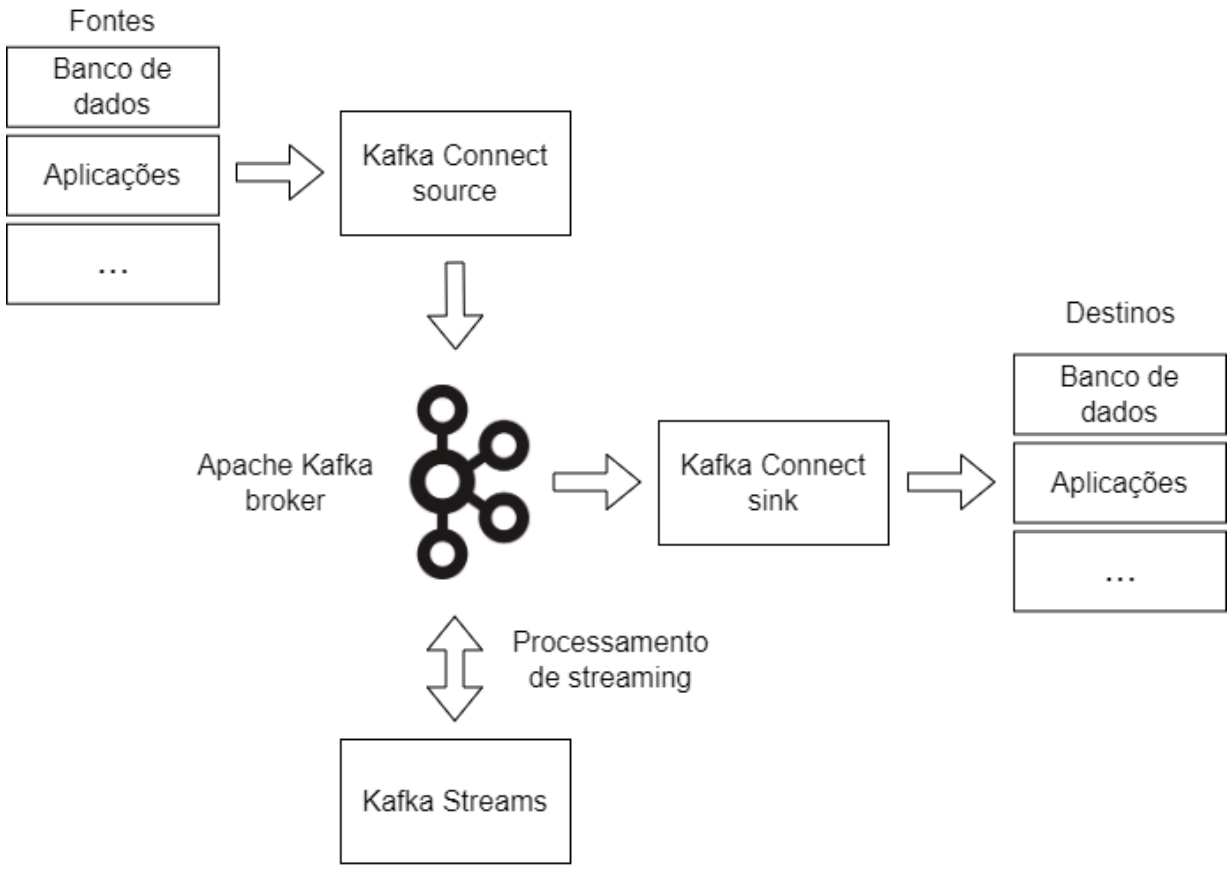

Figura 8.5: Ecossistema Apache Kafka para processamento de streaming de eventos.

A imagem a seguir mostra a correlação dos componentes da plataforma de *streaming* de eventos Apache Kafka com as etapas desse tipo de plataforma:

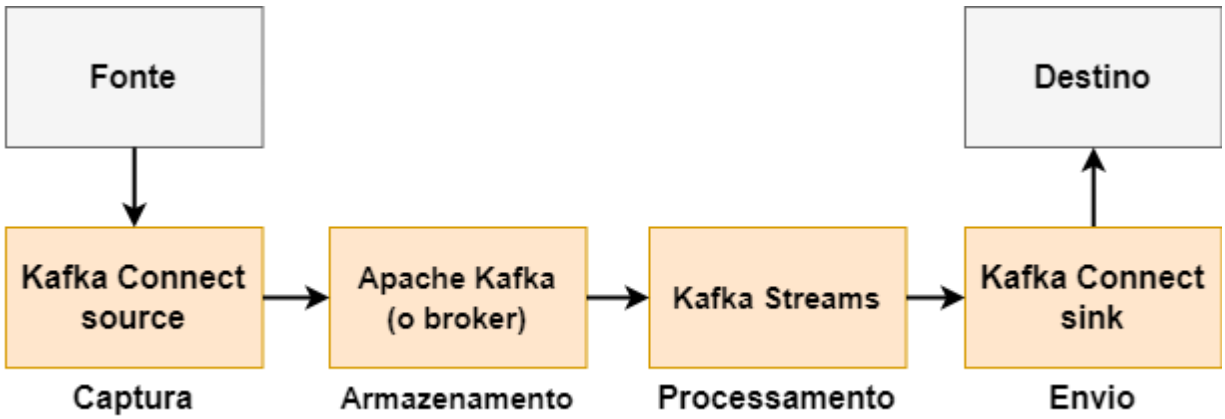

Figura 8.6: Correlação dos componentes da plataforma de streaming de eventos Apache Kafka com as etapas.

Para enriquecer a plataforma do Apache Kafka, a Confluent adicionou dois novos componentes: o *ksqlDB*, para processar fluxo de dados usando comando SQL, e o *registro de esquema*, para o gerenciamento de esquema, como vimos na seção *Gerenciamento de esquema* do capítulo 5, *Broker de eventos habilita a Arquitetura orientada a eventos*.

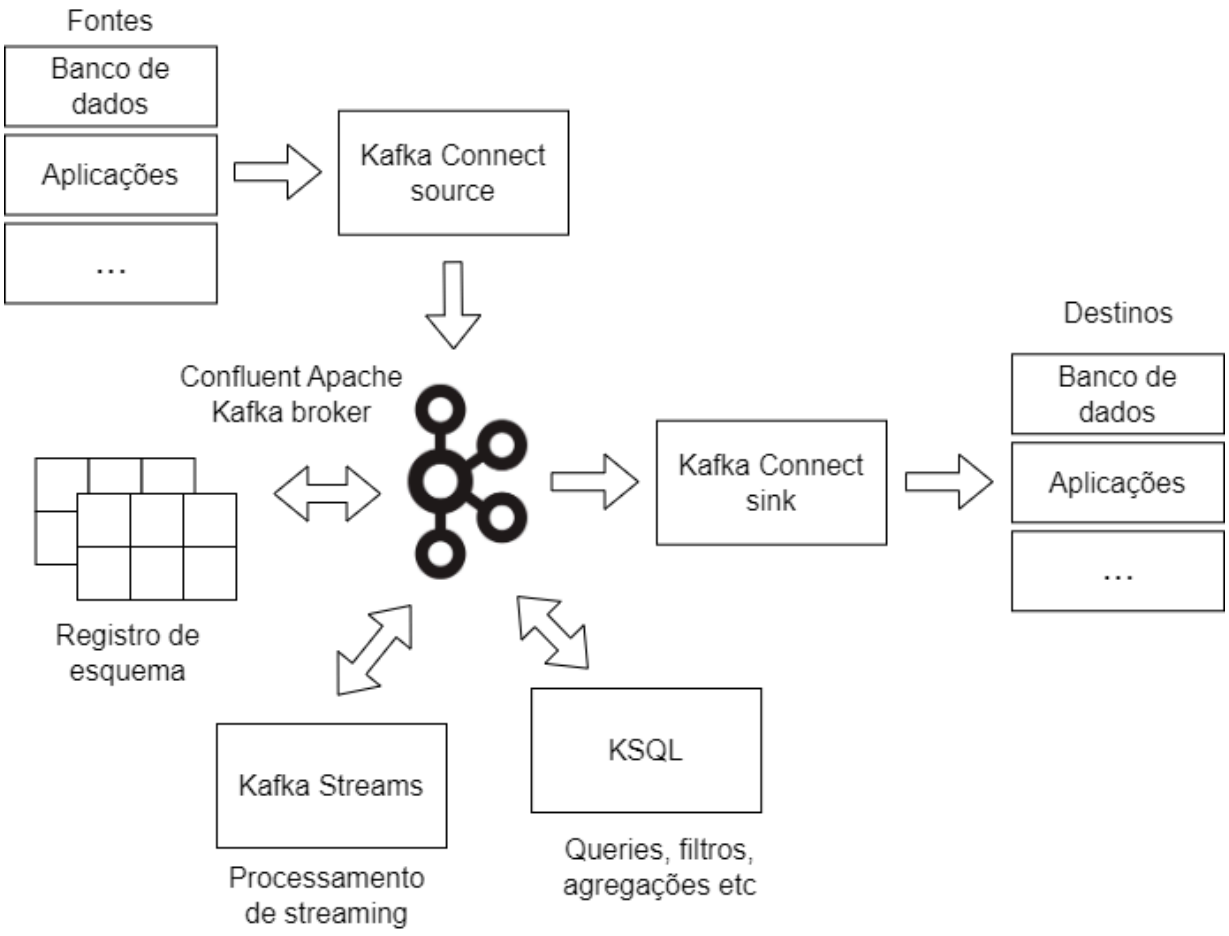

Figura 8.7: Ecossistema Confluent para processamento de streaming de eventos.

Além da plataforma do Apache Kafka, temos outras soluções de plataformas disponíveis no mercado, como Apache Flink (https://flink.apache.org/), Apache Spark (https://spark.apache.org/), Apache Storm (https://storm.apache.org/), Apache Pinot (https://pinot.apache.org/), entre outras.

Os provedores de nuvem não ficam de fora dessa. Eles também têm soluções para processamento de *streaming* de eventos, como Amazon Kinesis (https://aws.amazon.com/pt/kinesis/), Microsoft Event Hubs (https://azure.microsoft.com/pt-br/products/event-hubs), entre outras. Muitas dessas plataformas utilizam o *broker* de eventos do Apache Kafka para a etapa de armazenamento.

Segundo a Gartner em sua pesquisa *How to Identify Your Event-Driven Architecture Use Cases to Select the Best-Fit Event Broker* (2021b), os fornecedores de plataformas de *streaming* de eventos respondem por uma participação de aproximadamente 20% do mercado de *broker* de eventos, mas essa participação no mercado está crescendo rapidamente.

Concluindo

Com a necessidade de inovar mais rapidamente para atender a demandas de negócio cada vez mais desafiadoras, com uma avalanche de dados para processar, empresas que executam bem suas soluções digitais baseadas na EDA – em conjunto com pelo menos uma das arquiteturas que vimos neste capítulo – são empresas com soluções modernas, resilientes e que podem crescer de acordo com a demanda dos seus clientes.

Neste capítulo, tivemos uma visão geral das arquiteturas de microsserviços, *serverless* e de *streaming* de dados ou eventos de série e entendemos como elas utilizam e se beneficiam da EDA.

Apesar dos benefícios gerados por esses padrões arquiteturais do modelo distribuído, vale ressaltar que cada projeto de TI é único e que não existe uma solução única para todos os problemas. A decisão de adotar uma solução distribuída ou centralizada (sistema monolítico) depende de vários fatores, tendo sempre em mente os requisitos do negócio. Definir os padrões arquiteturais envolve avaliação (*trade-off*) das vantagens e desvantagens do padrão a ser adotado.

No próximo capítulo, vamos explorar como a EDA pode ser utilizada dentro de um contexto de plataforma de integração.

CAPÍTULO 9
Arquitetura orientada a eventos no contexto de uma plataforma de integração

A necessidade de integrar sistemas não é nova. Em um mercado cada vez mais competitivo, criar um único sistema para rodar todo o negócio não é mais viável. As empresas geralmente são compostas por centenas, até mesmo milhares, de sistemas e aplicações. Além disso, com tantas soluções disponíveis no mercado, podemos selecionar diversas soluções de fornecedores que melhor atendam aos requisitos de negócio do cliente. Por exemplo, queremos selecionar a melhor solução de atendimento ao cliente, a melhor solução de vendas, a melhor solução de planejamento de recursos empresariais (ERP), a melhor solução de pagamento, a melhor solução de *marketplace* e por aí vai.

Os clientes que interagem com a solução digital da empresa não querem saber quais sistemas são utilizados internamente no processo de negócio. O que eles querem é uma solução com uma experiência simples e unificada para atingir seus objetivos com rapidez. Imagine um cliente ter que acessar 10 sistemas distintos na sua jornada para atingir seu objetivo!

Integração de sistemas significa conectar sistemas, parceiros, fornecedores, clientes e compartilhar informações/dados — e uma plataforma de integração nos ajuda nessa missão. Neste último capítulo da parte *Arquitetura orientada a eventos e além*, vamos entender o que é uma plataforma de integração, suas principais capacidades e sua relação com a EDA.

9.1 O que é uma plataforma de integração

Plataforma de integração é uma combinação de capacidades com objetivo de suportar uma gama crescente de casos de uso relacionados à integração de sistemas. As principais capacidades estão relacionadas a **API**, **evento** e **fluxo de integração**, em conjunto com capacidades que as suportam, como *monitoração*, *segurança*, *observabilidade*, *operação* e *governança*.

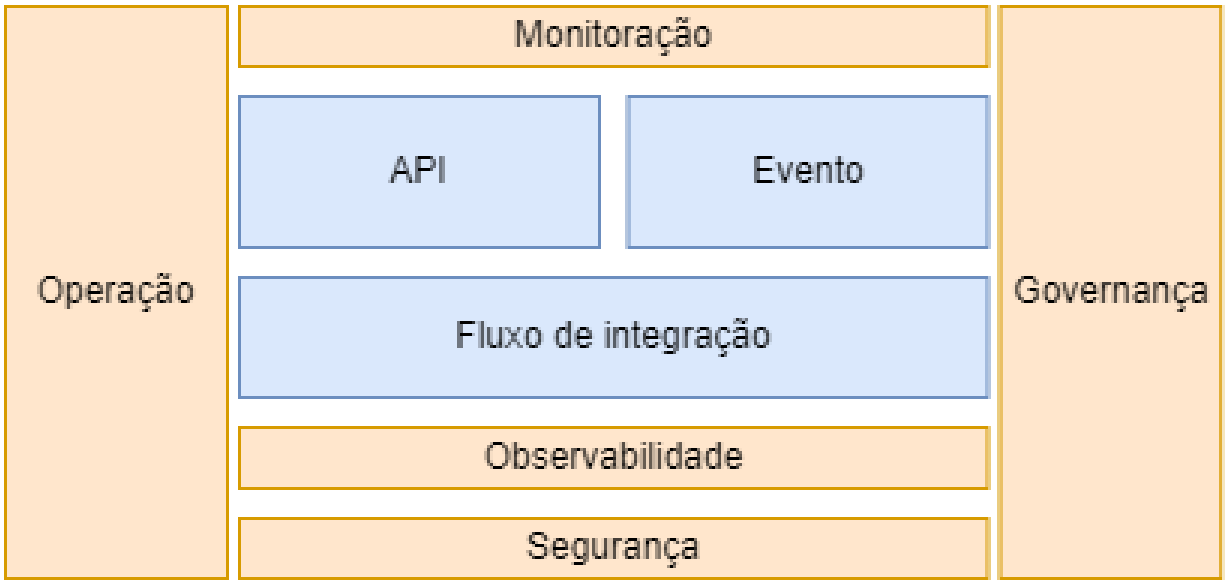

Figura 9.1: Principais capacidades de uma plataforma de integração.

Alguns fornecedores de plataformas de integração tentam dar uma experiência única dessas capacidades, fazendo com que seus clientes tenham uma única solução para resolver todos (ou quase todos) os casos de uso de integração. Para citar alguns:

- Mulesoft: https://www.mulesoft.com/platform/enterprise-integration
- Sensedia: https://www.sensedia.com/
- Tibco: https://www.tibco.com/products/cloud-integration
- WSO2: https://wso2.com/

Alguns exemplos de casos de uso comuns endereçados por essas plataformas são:

- Integrar aplicações para uma jornada do cliente mais simples e eficiente;
- Modernizar sistemas legados — por exemplo, migrar dados de sistemas legados para soluções modernas na nuvem;
- Integrar aplicações SaaS — por exemplo, solução de atendimento ao cliente (Zendesk) com aplicativo de mensagens instantâneas (Slack);
- Sincronizar dados entre soluções empresariais como SAP e Salesforce;
- Consolidar dados de fontes distintas em uma única visualização;
- Possibilitar uma experiência *omnichannel*, entre outros.

Nas próximas seções, vamos explorar o papel de um gerenciamento de API, a plataforma de integração como serviço e como a EDA se encaixa nesse contexto.

9.2 O papel do gerenciamento de API

O **gerenciamento de API** (do inglês, *API Management* ou simplesmente APIM), como o nome sugere, está relacionado à API dentro das principais capacidades de uma plataforma de integração.

O uso de APIs continua sendo muito popular na integração de sistemas e parceiros, bem como para expor e comercializar funcionalidades de negócio e/ou informações/dados. Sua utilização tem crescido ano após ano para atender ao crescimento do negócio e à agilidade tecnológica exigida pelos clientes. Essas APIs permitem que as empresas exponham suas funcionalidades de negócios internamente (para consumidores internos dentro dos seus domínios), externamente (para consumidores externos dos seus domínios) ou um pouco dos dois (quando precisa expor funcionalidades para consumidores parceiros).

Com essa crescente demanda do uso de APIs pelas empresas, veio a necessidade de gerenciá-las de uma maneira centralizada e segura. O APIM é um processo que gerencia todo o ciclo de vida da API.

O ciclo de vida de uma API corresponde ao:

1. Planejamento e design;
2. Implementação e teste;
3. Implantação (*deploy*) e operação;
4. Versionamento e aposentadoria.

Para contemplar o processo que gerencia todo o ciclo de vida da API de uma maneira centralizada e segura, fornecedores de plataformas de integração oferecem uma plataforma de gerenciamento de API. Essa plataforma tem como principais componentes:

1. *Gateway* de API (*API Gateway*);
2. Gerenciador de API (*API Manager*);
3. Portal do desenvolvedor (*Developer Portal*).

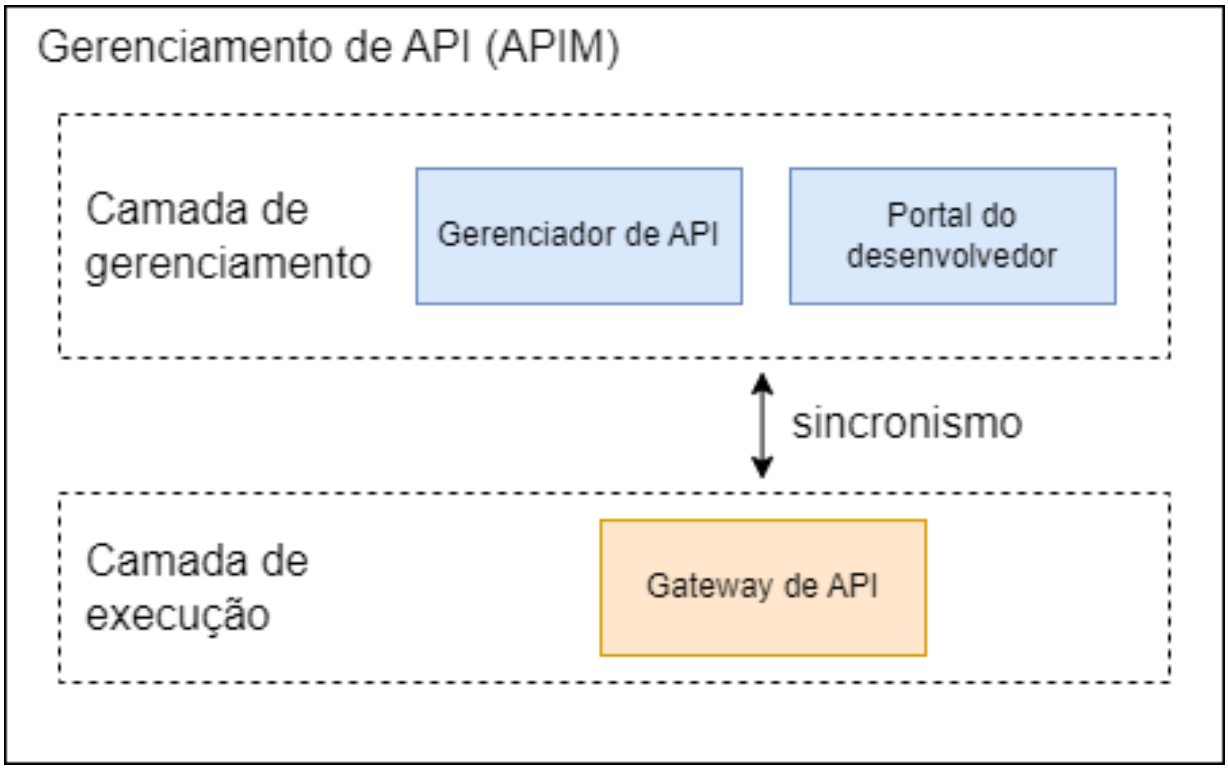

Figura 9.2: Gerenciamento de API – camada de gerenciamento e execução.

Gateway de API

O ***gateway* de API** é o componente mais importante de um APIM. Ele faz parte da camada de execução. Ele é o ponto de entrada de todas as solicitações (*requests*) das aplicações clientes (consumidores das APIs) para os serviços de *back-end*, onde o negócio é efetivamente executado.

O *gateway* garante que cada solicitação a uma API passe por um conjunto de políticas pré-configuradas até chegar nos serviços de *back-end*, seja microsserviços, funções *serverless* ou até mesmo um *broker* de eventos.

Suas principais responsabilidades são:

- Validar a solicitação de acordo com o contrato definido pela API;
- Aplicar autenticação e política de segurança, integrado a um provedor de identidade;

- Controlar a taxa de solicitações de uma aplicação cliente em um determinado período (*Rate limiting*);
- Aplicar política de cache para melhorar o tempo de resposta para a aplicação cliente e não sobrecarregar os serviços de *back-end*;
- Aplicar roteamento dinâmico para os serviços de *back-end*;
- Aplicar tradução de protocolo. Por exemplo, as aplicações clientes utilizam `HTTPS` para se comunicar com o *gateway* e os serviços de *back-end* utilizam `gRPC` ou qualquer outro protocolo.

A imagem a seguir é um exemplo de referência de um fluxo de solicitações utilizando um *gateway* de API entre as aplicações clientes e os serviços de *back-end*.

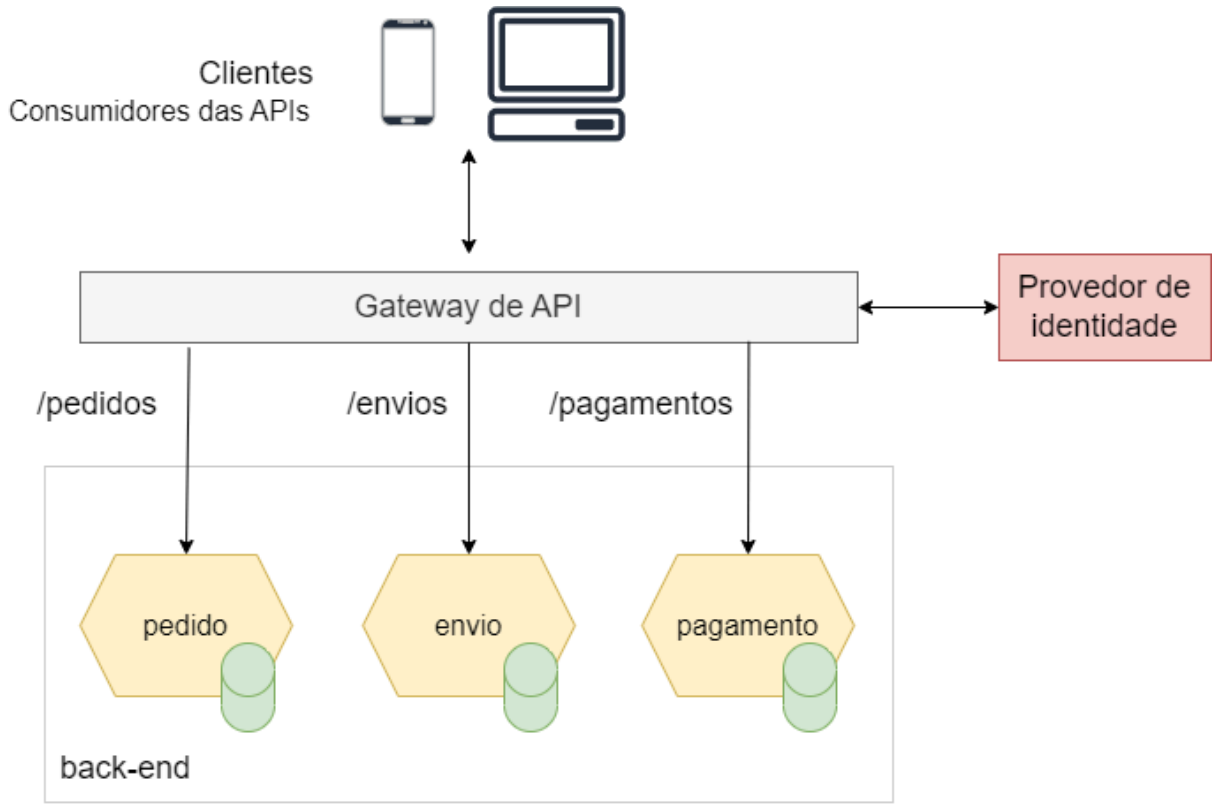

Figura 9.3: Exemplo de um fluxo típico de solicitações utilizando um gateway de API.

Todas as políticas que o *gateway* de API executa são configuradas pela pessoa desenvolvedora da API no gerenciador de API, que veremos a seguir.

Gerenciador de API

O **gerenciador de API** faz parte da camada de gerenciamento do APIM. É nele que a pessoa desenvolvedora da API projeta, cria e publica suas APIs no *gateway*. Além disso, o(a) desenvolvedor(a) define os contratos das API e configura as políticas de segurança. Também é possível fazer composição de APIs (agrupar o resultado de várias APIs para dar uma única resposta para a aplicação cliente), além de monetizar, monitorar e governar todo o catálogo de APIs.

Tudo que é criado e alterado pelos desenvolvedores da API no gerenciador da API é automaticamente aplicado na camada de execução da API, isto é, no *gateway* da API. Todo o tráfego de uso das APIs e métricas geradas pelo *gateway* são sincronizadas com a camada de gerenciamento. Isso permite que os clientes acompanhem o uso das suas APIs através de *dashboards* e façam análise das métricas técnicas e de negócio.

A camada de gerenciamento, também conhecida como *control plane*, é totalmente desacoplada da camada de execução, também conhecida como *data plane*. Se a camada de gerenciamento cair, o *gateway* continua funcionando, mas as informações de execução vão ter atraso de sincronismo nos *dashboards* e relatórios diversos.

A imagem a seguir é um *screenshot* do gerenciador de API da Sensedia.

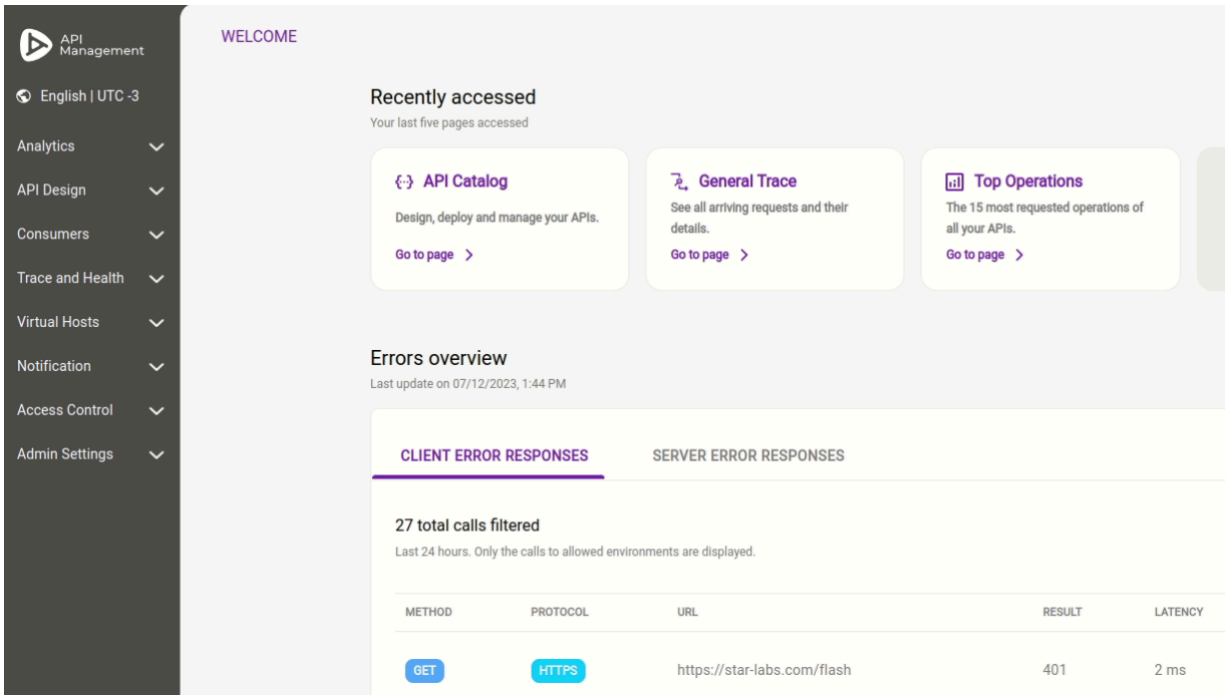

Figura 9.4: Screenshot da tela principal do gerenciador de API da Sensedia.

Portal do desenvolvedor

Assim como o gerenciador de API, o portal do desenvolvedor também faz parte da camada de gerenciamento do APIM.

O **portal do desenvolvedor** é um site personalizado para o cliente com toda a documentação das suas APIs. Seu principal objetivo é prover um canal de autoatendimento para os desenvolvedores das aplicações clientes acessarem as APIs. Com ele, os consumidores das APIs conseguem criar uma conta, adquirir credenciais, utilizar SDKs, descobrir as APIs através do catálogo, testar as APIs, entre outras funcionalidades.

A seguir, a lista de alguns portais públicos de desenvolvedores:

- Cielo: https://desenvolvedores.cielo.com.br/api-portal/
- Elo: https://dev.elo.com.br/
- MAG: https://developers.mag.com.br/api-portal/;
- Meta: https://developers.facebook.com/

- LinkedIn: https://developer.linkedin.com/

9.3 O papel da plataforma de integração como serviço

A plataforma de integração como serviço (do inglês *Integration Platform As A Service*, ou simplesmente iPaaS), **está relacionada a fluxo de integração** dentro das principais capacidades de uma plataforma de integração.

O **iPaaS** é uma plataforma que nos permite conectar diferentes tipos de aplicações, negócios, dispositivos IoT e dados sem instalar ou gerenciar hardware. Ele é nativamente baseado na nuvem, mas pode ser executado na infraestrutura do cliente (*on-premise*).

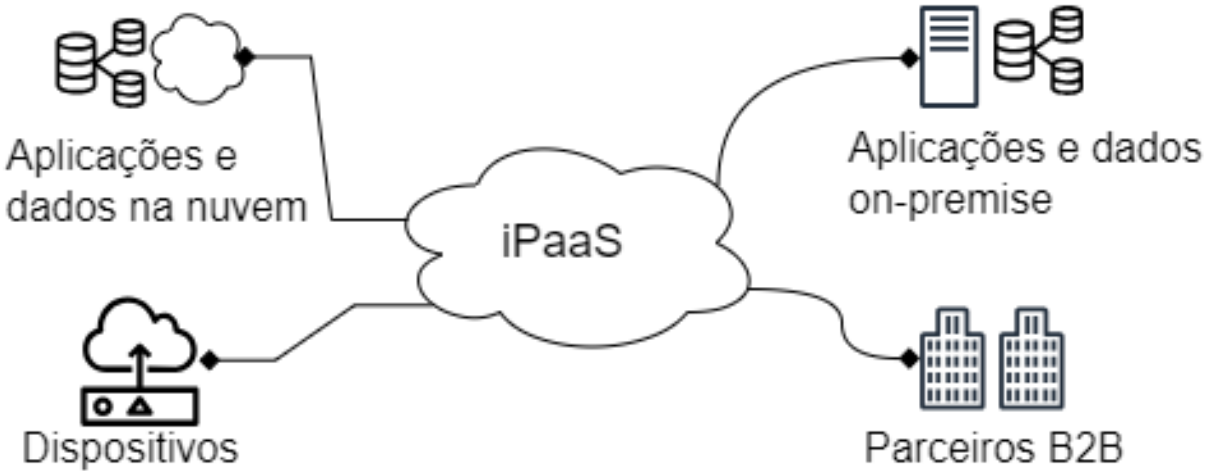

Figura 9.5: Plataforma de integração como serviço (iPaaS).

Com uma solução de iPaaS, é possível desenvolver, implantar, executar e gerenciar fluxos de integrações complexas de forma centralizada e com pouca codificação (*low-code*), acelerando a automação de processos de negócio e compartilhamento de dados entre sistemas.

Qualquer empresa que tem produto no modelo *Software as a Service* (SaaS) vende seu software para seus clientes com uma assinatura, normalmente paga mensalmente. Junto com a assinatura, essas empresas oferecem outros serviços para compor a oferta, como suporte técnico, serviços de consultoria, funcionalidades como complementos (*add-ons*) para estender a oferta e por aí vai.

Com *iPaaS* é o mesmo princípio, só que referente a uma solução de plataforma de integração (que também é um software, no final das contas).

Para atender à necessidade de integração de sistemas das empresas, uma solução de iPaaS é composta por um conjunto de componentes. Os principais componentes são:

1. Ambiente de desenvolvimento *low-code*;
2. Acionadores (*Triggers*);
3. Conectores (*Connectors*);
4. Mapeamento e transformação de dados (*Data mapping*).

Os principais componentes são suportados por capacidades como monitoração, segurança, observabilidade, operação e governança.

Ambiente de desenvolvimento low-code

Ambiente de desenvolvimento *low-code* de uma solução de *iPaaS* significa que conseguimos criar um fluxo de integração com pouca codificação para integrar sistemas.

O desenvolvedor monta todo o fluxo de integração arrastando e soltando componentes pré-fabricados e os conectando em uma sequência no Canvas. Esses componentes são de fácil configuração e customização para atender aos requisitos de negócio do cliente. Isso habilita a empresa a criar e atualizar seus fluxos de integração muito rápido, permitindo-lhe acompanhar as demandas de negócios em constante mudança.

Acionadores

Os **acionadores** ou **gatilhos** (*triggers*) são o ponto de partida do fluxo de integração. Quando acionado, esse componente inicia a execução do fluxo. Por exemplo, podemos iniciar um fluxo de integração a partir de um *endpoint* REST/HTTPS ou *webhook*; a partir de um evento; a partir de um agendador (*scheduler*); a partir de uma alteração em um arquivo etc.

Os fornecedores de iPaaS oferecem uma biblioteca de acionadores pré-fabricados. Por exemplo, muitas aplicações SaaS oferecem integração via *webhook*, como vimos na seção *Webhooks* do capítulo 3, *Padrões de arquitetura para a Arquitetura orientada a eventos*. Os acionadores pré-fabricados do tipo *webhook* já vêm com toda a estrutura preparada de acordo com a documentação oferecida pela aplicação SaaS, basta apenas o cliente do iPaaS ter uma assinatura válida da aplicação SaaS e fazer as devidas configurações no acionador.

Conectores

Os **conectores** são os componentes-chaves de um iPaaS. Eles são responsáveis pela comunicação com os sistemas externos.

O que diferencia um fornecedor de iPaaS é sua biblioteca de conectores pré-fabricados. Os conectores fazem com que o cliente da plataforma consiga fazer integração com sistemas externos com poucas configurações.

Podemos dividir os conectores em três tipos:

1. **Conectores de negócio ou de aplicação**: são conectores específicos de soluções/aplicações de mercado. Por exemplo, conector para SalesForce, SAP, Zendesk, AWS S3, Hubspot, Slack, entre outros;
2. **Conectores de fonte de dados**: são conectores de banco de dados SQL e NoSQL. Por exemplo, Oracle, MySQL, PostgreSQL,

MongoDB, Amazon DynamoDB, entre outros;
3. **Conectores de tecnologia**: conectores de tecnologia são especializados na tecnologia, mais precisamente no protocolo de comunicação. Por exemplo, HTTP, FTP, AMQP, protocolo do Kafka (*wire protocol*), TCP/IP, entre outros.

A tecnologia é abstraída nos conectores de negócio. Em cada conector de negócio, é utilizado um protocolo de comunicação. Por mais que um fornecedor de iPaaS possa ter um rico portfólio de conectores pré-fabricados, nunca será o suficiente. Por isso, muitos fornecedores têm conectores de tecnologia, para possibilitar os clientes a usarem de forma genérica ou até mesmo a desenvolverem o seu próprio conector através do kit de desenvolvimento (SDK).

Mapeamento e transformação de dados

Cada sistema envolvido na integração tem seu protocolo de comunicação e sua estrutura de dados. O componente de mapeamento e transformação de dados tem a habilidade de mapear (*map*), transformar (*transform*), agregar (*aggregate*) e dividir (*split*) os dados de acordo com a estrutura de dados do sistema de origem para a estrutura de dados do sistema destino.

Além disso, ele contempla funções pré-fabricadas para acelerar o desenvolvimento. Por exemplo, funções de manipulação de *strings*, como concatenação, *split*, formatação, funções de data, como gerar data atual, somar datas etc.

Esse componente pode ser um componente à parte no fluxo de integração e/ou pode estar embutido dentro dos conectores. Os conectores, além de abstraírem a comunicação com o protocolo, carregam a estrutura de dados automaticamente de acordo com o que o sistema destino espera receber. E, dentro do conector, o cliente realiza o mapeamento de dados, também conhecido como *de-para*.

O exemplo de mapeamento de dados da imagem a seguir mostra uma estrutura de dados do sistema origem (`Usuario`) sendo mapeada/transformada para uma estrutura de dados do sistema destino (`Contato`). Cada sistema tem sua estrutura. Note que, na estrutura de dados do *sistema destino*, não existe um campo de sobrenome: o sistema destino espera receber o nome completo. Neste caso, uma concatenação foi realizada através de uma função pré-fabricada.

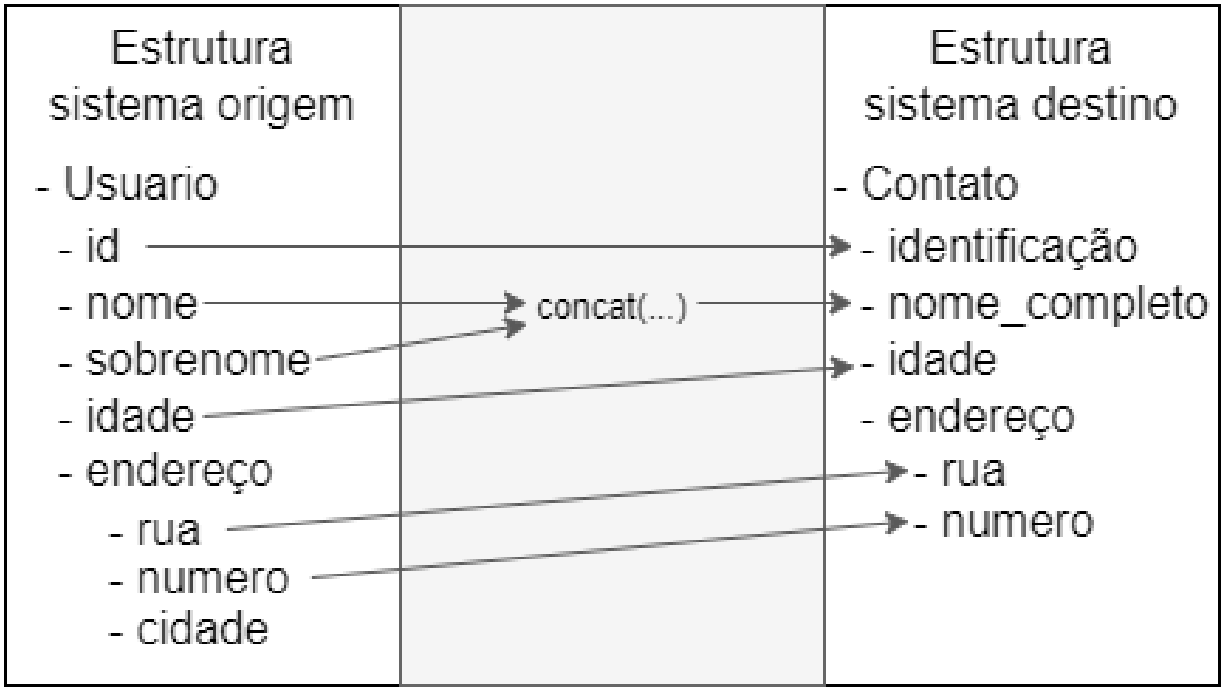

Figura 9.6: Exemplo de um componente de mapeamento de dados low-code.

Com o objetivo de exemplificar o uso de uma solução de iPaaS, vamos simular a criação de um fluxo de integração para a empresa XYZ. Imagine que a empresa XYZ tem duas soluções/aplicações SaaS: o HubSpot (https://www.hubspot.com/), sua solução de vendas, e o SalesForce (https://www.salesforce.com/br/), sua solução de relacionamento com clientes.

A cada negócio (*deal*) criado na solução de vendas do Hubspot, a empresa XYZ precisa que seja criada uma oportunidade (*opportunity*) na sua solução de relacionamento de cliente do

SalesForce. Caso o valor do negócio seja maior do que R$ 50 mil, a empresa XYZ precisa que o gerente da conta responsável seja notificado via Slack.

A imagem a seguir, é um exemplo típico de um fluxo de integração de um iPaaS *low-code* para atender à necessidade da empresa XYZ.

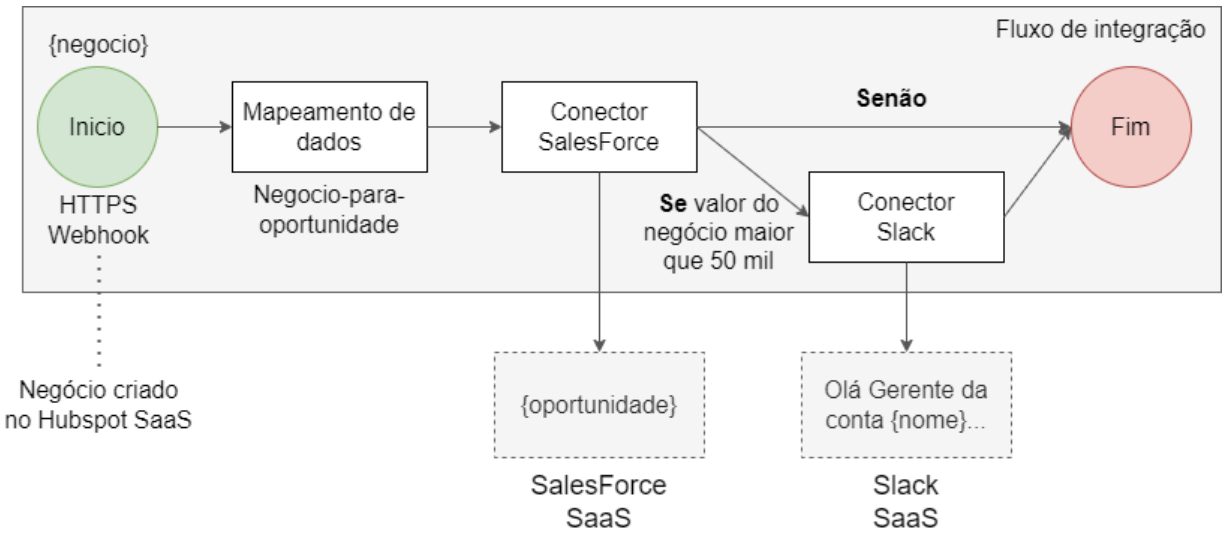

Figura 9.7: Exemplo típico de um fluxo de integração de um iPaaS low-code.

- O fluxo de integração é acionado quando um negócio (*deal*) é criado no Hubspot via *webhook*;
- O primeiro mapeamento de dados faz a transformação da estrutura de dados do Hubspot (negócio) para a estrutura de dados do SalesForce (oportunidade);
- O fluxo de integração faz a comunicação com o SalesForce através do conector para criar a oportunidade;
- Após a criação da oportunidade no SalesForce, o fluxo de integração verifica se o valor do negócio é maior do que 50 mil:
    - Se for maior do que 50 mil, então o fluxo de integração faz a comunicação com o Slack através do conector para notificar o gerente da conta responsável e finaliza o fluxo;
    - Se não for maior do que 50 mil, o fluxo de integração é finalizado sem notificação via Slack.

9.4 Arquitetura orientada a eventos e a plataforma de integração

Como você pode imaginar, a EDA está relacionada a *eventos* dentro das principais capacidades de uma plataforma de integração.

Uma plataforma de integração utiliza a EDA para habilitar sistemas a sincronizar seus dados em tempo real, além, obviamente, de desacoplá-los e garantir que as informações/dados do sistema origem ou fonte (publicador) sejam sincronizadas com os sistemas destinos ou alvos (consumidores) através da sua capacidade de garantia de entrega, que vimos na seção *Garantia de entrega da mensagem* do capítulo 2, *Entendendo os principais conceitos*.

Imagine uma solução de *e-commerce* na qual recebemos uma solicitação de pedido do cliente pelo *gateway* de API, que roteia para um microsserviço para executar a regra de negócio. Após o processamento da regra de negócio, o microsserviço publica um evento de `pedido-criado` no *broker*. O evento `pedido-criado` aciona um fluxo de integração complexo do iPaaS para sincronizar o pedido recém-criado com diversos sistemas externos — tudo isso sendo monitorado, rastreado e governado por uma única plataforma de integração, contemplando API, eventos e fluxo de integração.

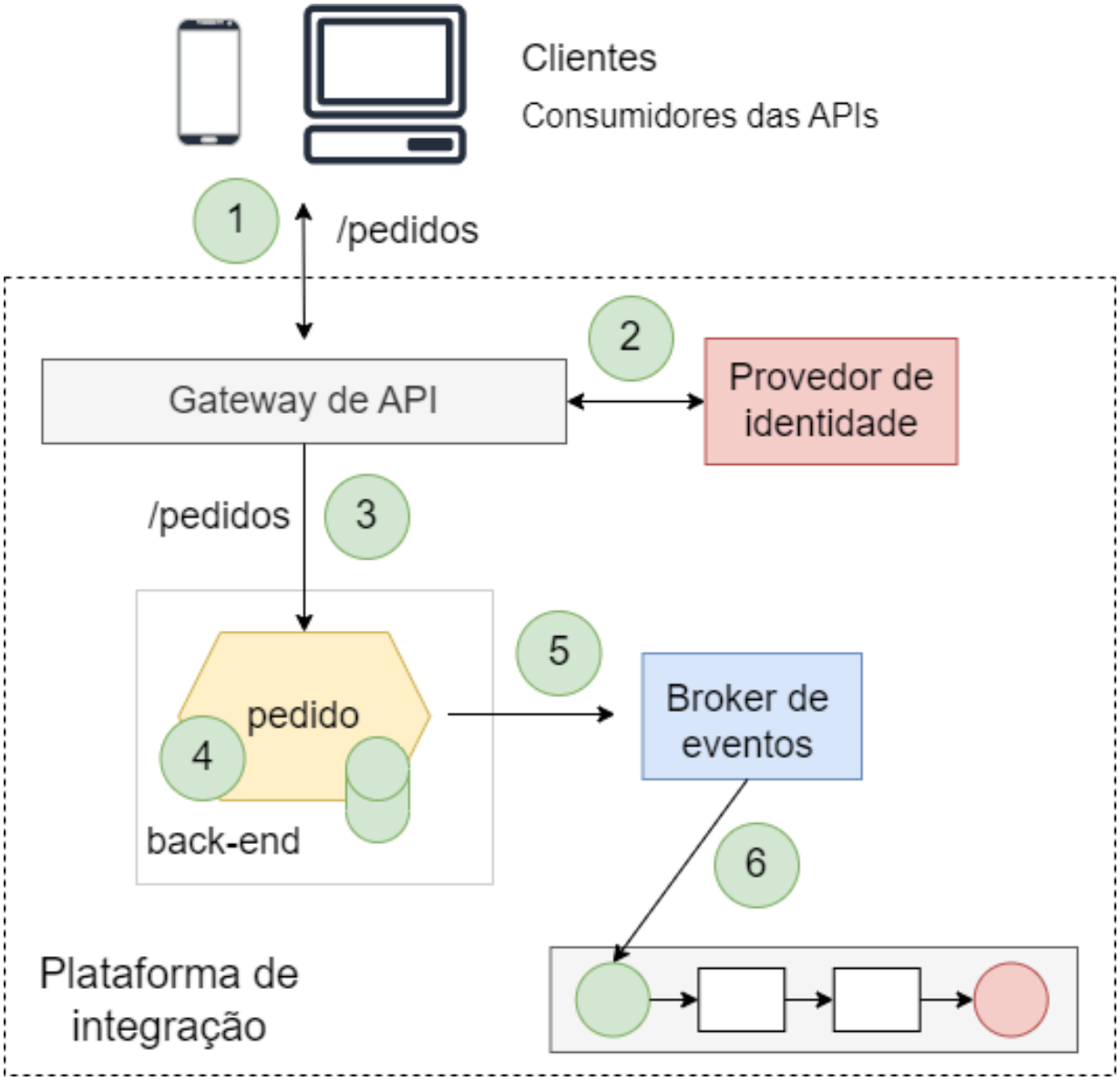

Figura 9.8: Exemplo de solução com Gateway de API, microsserviço, eventos e iPaaS.

1. O cliente solicita um pedido através do aplicativo (aplicação cliente);
2. A aplicação cliente é autenticada e autorizada no provedor de identidade;
3. O *gateway* da API faz o roteamento para o microsserviço `pedido` ;
4. O microsserviço `pedido` executa a regra de negócio;

5. Após o processamento da regra de negócio, o microsserviço `pedido` publica o evento `pedido-criado` no *broker*;
6. O fluxo de integração é acionado com o evento `pedido-criado`.

Concluindo

Com a demanda crescente de integração de sistemas, uma solução de plataforma de integração vem a calhar. A Gartner estima, em sua pesquisa *Magic Quadrant for Enterprise Integration Platform as a Service* (THOO *et al.*, 2021), que o mercado de plataforma de integração gerou 3,47 bilhões de dólares de receita em 2020 e cresceu 38,7% em comparação com 2019. Ela espera que o mercado de plataforma de integração excederá 9 bilhões de dólares em receita até 2025.

Neste capítulo, exploramos as principais capacidades de uma plataforma de integração, que estão relacionadas a **API**, **evento** e **fluxo de integração**. Entendemos o papel do gerenciamento de API na plataforma, que está relacionado à capacidade de API. Vimos também o papel de um iPaaS, que está relacionado à capacidade de fluxo de integração, e, na sequência, entendemos como uma plataforma de integração se beneficia da EDA.

No próximo capítulo, começa a parte IV do livro, na qual vamos desenvolver alguns projetos de EDA para colocar em prática o que aprendemos.

Parte IV: Execução dos projetos de EDA

Capítulo 10
Projeto 1 – Plataforma de e-commerce

Bem-vindo(a) à última parte do livro!

Chegamos até aqui bem equipados com os principais conceitos e padrões de EDA (parte I), com as principais características e funcionalidades de um *broker* de eventos (parte II) e, para finalizar, com as principais arquiteturas e plataformas utilizadas em diversos projetos de TI que têm uma relação muito próxima com a EDA (parte III).

Nesta parte final, vamos executar três projetos fictícios e simplificados para colocar em prática o que aprendemos. A ideia é, para cada projeto, termos a definição:

1. Do problema e do escopo;
2. Dos requisitos funcionais (*functional requirements*);
3. Dos requisitos não funcionais (*non-functional requirements*);
4. Das restrições (*constraints*).

Com base nessas definições, entramos na fase de execução do projeto. A fase da execução está dividida em quatro etapas:

1. Escolha do *broker* de eventos e do padrão arquitetural;
2. Design da solução;
3. Teste;
4. Implementação.

A etapa de implementação é onde você terá a oportunidade de colocar a mão na massa!

Em cada projeto, executaremos as etapas de **escolha do *broker* de eventos e do padrão arquitetural**, **design da solução** e **teste** com informações valiosas para que você possa implementar a solução proposta por conta própria. Você poderá testar sua implementação utilizando a coleção do *postman* que está no nosso repositório do GitHub, além de poder comparar a sua implementação com a solução lá disponível. Ao final de cada capítulo, o leitor e a leitora serão desafiados com sugestões extras para que possam enriquecer os artefatos gerados nas etapas de **design da solução** e **teste** para evoluírem suas implementações.

Escolha do broker de eventos e do padrão arquitetural

Nesta etapa, vamos escolher o *broker* de eventos e o padrão arquitetural de acordo com o problema, os requisitos e as possíveis restrições. Com a definição do *broker* de eventos e o padrão arquitetural, podemos avançar para as próximas etapas.

Design da solução

Na etapa de design da solução, vamos gerar os artefatos necessários para que você possa implementar a solução. Esses artefatos são os arquivos de *OpenAPI* e *AsyncAPI*, bem como o diagrama da arquitetura da solução, o diagrama da coreografia de eventos, entre outros diagramas de apoio. Cada projeto tem suas particularidades e necessidades. Os tipos de diagramas podem variar de acordo com o projeto.

Teste

Na etapa de teste, vamos definir o cenário de teste principal do caminho feliz, utilizando o modelo de **desenvolvimento orientado a comportamento** – do inglês, *Behaviour-Driven Development* (BDD) –, e automatizá-lo através do *postman*.

Implementação

Nesta etapa, você vai implementar a solução de acordo com os artefatos gerados na etapa de design da solução e de acordo com o cenário de teste principal do caminho feliz, criado na etapa de teste. A etapa de implementação contempla a implantação da solução (*deploy*).

A imagem a seguir resume as etapas que vamos executar em nossos projetos:

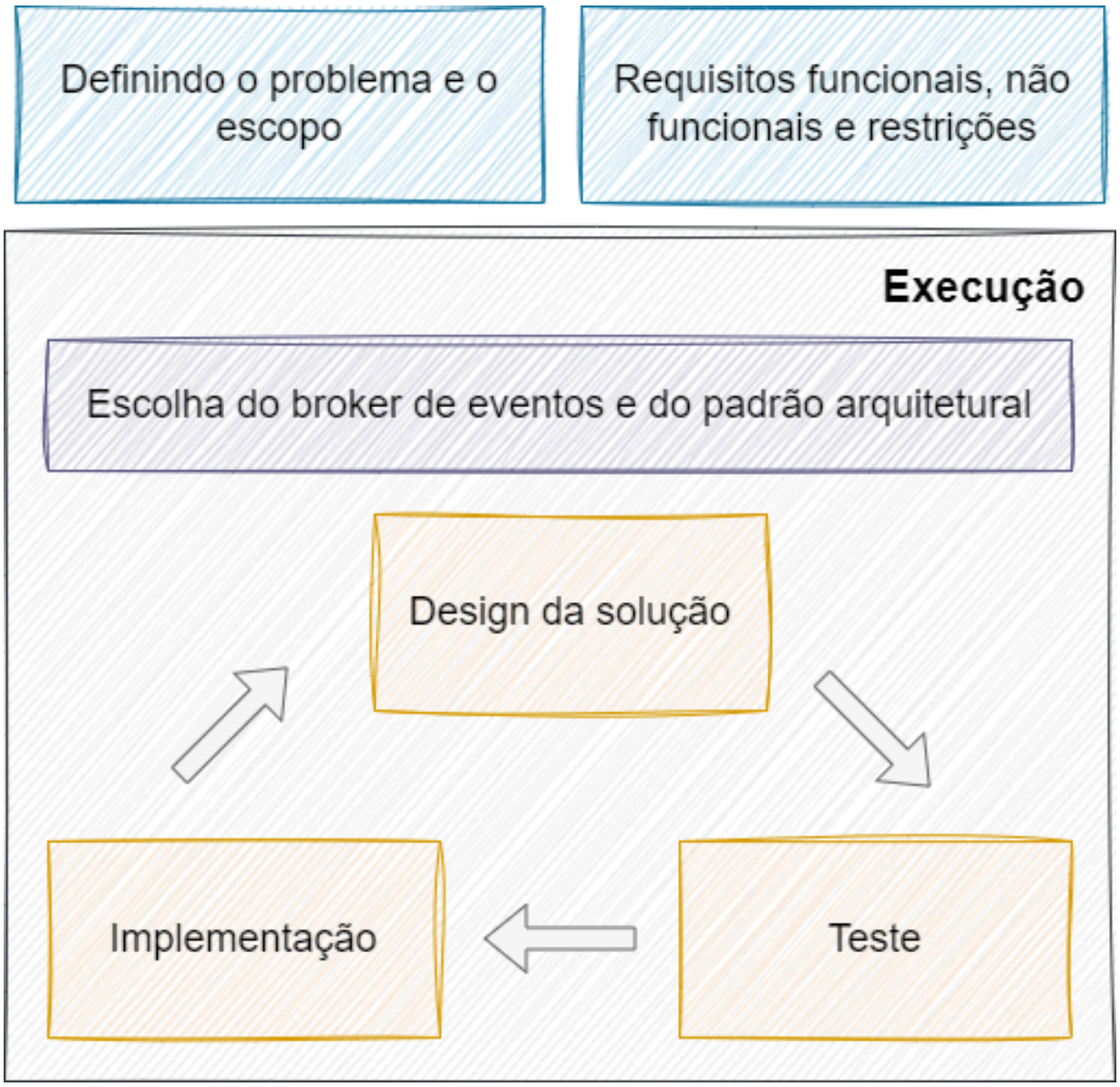

Figura 10.1: Etapas dos projetos.

Os artefatos gerados nas etapas de *design da solução*, *teste* e *implementação* estão disponíveis no nosso projeto do GitHub.

A partir daqui, iniciamos o nosso primeiro projeto.

10.1 Entendendo o problema e definindo o escopo

A empresa ABC está com um grande problema: a sua solução de e-commerce não consegue atender ao crescimento da demanda de pedidos dos seus clientes, o que está fazendo com que a empresa perca receita/venda e credibilidade.

A pandemia da COVID-19 alavancou o negócio da empresa e surpreendeu a todos com o crescimento repentino. A empresa passou de 500 pedidos processados por dia para 5 mil pedidos processados por dia, com estimativa de crescimento de 30% no próximo ano.

A empresa ABC precisa garantir que todos seus clientes consigam realizar suas compras em qualquer um dos seus canais digitais. Sua solução de e-commerce precisa estar preparada para atender à demanda dos seus clientes.

O processo de venda da empresa ABC é simples e contempla os seguintes passos:

1. O cliente solicita o pedido por um dos canais digitais. O cliente precisa informar seu endereço de entrega e os dados de pagamento;
2. O sistema verifica se o endereço do cliente é elegível para entrega;
3. Caso o endereço não seja elegível, o sistema notifica o cliente por e-mail, informando que seu pedido foi cancelado e o motivo;

4. O sistema verifica se todos os produtos do pedido estão disponíveis no estoque;
5. Caso não haja pelo menos um produto em estoque, o sistema notifica o cliente por e-mail, informando que seu pedido foi cancelado e o motivo;
6. Caso haja todos os produtos em estoque e o endereço é elegível, o sistema realiza o pagamento através de um *gateway* de pagamento de terceiro;
7. Se o pagamento for autorizado, a empresa ABC inicia a preparação do pedido;
8. Após o pedido preparado, o sistema realiza o envio através de um serviço de entrega de terceiro e notifica o cliente por e-mail avisando que seu pedido foi enviado para seu endereço;
9. Uma vez o pedido entregue, o sistema é notificado e o pedido é finalizado com o status de entregue;
10. Caso o pagamento seja negado, a empresa ABC notifica o cliente por e-mail, informando que seu pedido foi cancelado e o motivo do pagamento não autorizado.

Qualquer erro que ocorra no processo, o pedido é enviado para uma central de operações para tratamento manual nesse primeiro momento.

10.2 Requisitos funcionais, não funcionais e restrições

Baseado no processo de negócio e nas inúmeras reuniões com o time de negócios da empresa ABC, chegamos ao consenso dos principais requisitos funcionais, não funcionais e das restrições impostas.

Requisitos funcionais

- O endereço e os dados de pagamento do cliente são informações obrigatórias;
- Apenas as capitais são elegíveis para entrega do pedido;
- O produto é considerado *em estoque* quando tem pelo menos três unidades;
- Os clientes podem consultar a situação dos seus pedidos.

Requisitos não funcionais

- *Alta disponibilidade e escalabilidade*: o sistema deve garantir que atenderá a todos os pedidos realizados pelos clientes de acordo com a previsão de crescimento;
- *Confiabilidade*: uma vez que o pedido foi recebido, ele não pode ser perdido, e pagamentos duplicados são inadmissíveis;
- *Consistência eventual*: as informações do pedido não precisam de consistência forte na consulta; ocorrer alguns segundos de atraso na consistência da informação do pedido é aceitável;
- *Extensibilidade da solução*: já é sabido que o processo de venda da empresa ABC será bastante evoluído, e para acompanhar essa mudança, o sistema deve ser capaz de receber novas funcionalidades com o menor impacto possível.

Restrições

- Por questões legais, toda a solução deve ser implantada e executada na infraestrutura do cliente (*on-premise*).

10.3 Etapa 1: Escolha do broker de eventos e das arquiteturas e plataformas

Nesta seção, vamos definir o *broker* de eventos e o padrão arquitetural de acordo com os requisitos do projeto através do registro de decisão arquitetural.

Registro de decisão arquitetural

O registro de decisão arquitetural, do inglês, *Architectural Decision Record* (ADR), é um documento com o racional das principais decisões arquiteturais realizadas para o projeto.

> O que eu vejo como uma grande vantagem do ADR é que ele mantém o histórico de todo o racional das decisões. A equipe do projeto consegue comunicar melhor as decisões tomadas para os novos membros da equipe.

Vamos criar o nosso ADR utilizando um template bem simples, com apenas o título, a decisão e a justificativa. Se você tiver interesse em utilizar o ADR em seu próximo projeto, inicie com um template simples — por exemplo, o proposto pelo Michael Nygard (https://adr.github.io/) ou de acordo com o guia proposto pela AWS (disponívle em https://docs.aws.amazon.com/pt_br/prescriptive-guidance/latest/architectural-decision-records/welcome.html).

Vamos criar dois registros de decisão arquitetural com a justificativa do *padrão arquitetural* e do *broker de eventos* que utilizaremos em nosso primeiro projeto.

Título

Padrão arquitetural.

Decisão

Vamos utilizar a arquitetura de microsserviço com comunicação assíncrona com EDA.

Justificativa

A arquitetura de microsserviço e a EDA possibilitam que a solução seja escalável e com alta disponibilidade para atender à demanda atual de 5 mil pedidos por dia e ao crescimento previsto para os próximos anos.

Essa decisão nos dá uma solução flexível e desacoplada para ser evoluída, possibilitando aos times adicionar novas funcionalidades e implantá-las (*deploy*) constantemente, sem precisar de longas datas programadas de implantação e, como consequência, fazendo a empresa ABC ser competitiva no mercado.

Como as informações do pedido não precisam de consistência forte na consulta, a consistência eventual através da EDA é aceitável.

Título

Broker de eventos para distribuição dos eventos.

Decisão

RabbitMQ 3.10.X. Um *broker* de eventos orientado a fila.

Justificativa

O *RabbitMQ* é um *broker* leve e de fácil implantação com suporte ao padrão de entrega *pub/sub* e a uma variedade de protocolos (*multiprotocolo*). A taxa de transferência (*throughput*) e a baixa latência (*low latency*) do RabbitMQ 3.10.X atende à nossa demanda de 5 mil pedidos por dia e ao crescimento previsto para os próximos anos. Nossa escolha foi feita com base no relatório de desempenho da versão 3.10 do RabbitMQ (https://blog.rabbitmq.com/posts/2022/05/rabbitmq-3.10-performance-improvements/).

O RabbitMQ tem suporte à semântica de *entrega pelo menos uma vez*. Com isso, todos os nossos microsserviços serão consumidores *idempotentes*. Isso garante o processamento de todos os pedidos e o não processamento do mesmo evento mais de uma vez, o que poderia ocasionar pagamentos duplicados, por exemplo.

A equipe do projeto é especialista em Java e utiliza o framework *Spring Boot 3.0*. Esse framework tem biblioteca com suporte ao RabbitMQ que acelera o desenvolvimento. Não temos como requisito reter os eventos/pedidos e muito menos reproduzi-los. O RabbitMQ remove os eventos após o consumo. Esse comportamento é o ideal para nosso caso de uso para evitarmos despesas com armazenamento de pedidos/eventos desnecessariamente.

Para atender à restrição imposta, que é a execução da solução na infraestrutura do cliente (*on-premise*), vamos utilizar o padrão de implantação do tipo *software* do RabbitMQ. Então vamos instalar e manter o *cluster* do RabbitMQ nos servidores da empresa.

10.4 Etapa 2: Design da solução

Nesta etapa, vamos gerar todos os artefatos necessários para que você possa implementar a solução:

1. Diagrama de contexto do sistema;
2. Diagrama da arquitetura da solução;
3. Diagrama da coreografia de eventos (*The Big Picture*);
4. Diagrama de máquina de estado do pedido;
5. Diagrama de sequência do caminho feliz da compra de um produto;
6. OpenAPI e AsyncAPI.

As versões finais de alguns artefatos ficaram muito grandes para serem impressas no livro. Nesses casos, vamos colocar uma versão resumida, com os principais componentes desses diagramas no livro para mantermos uma sequência padronizada e informar quando isso ocorrer. Todos os artefatos deste projeto estão disponíveis no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap10).

Diagrama de contexto do sistema

O objetivo do nosso primeiro diagrama é visualizar de forma macro os atores, pessoas ou papéis que interagem com o sistema e suas dependências externas. Ele é um bom ponto de partida para comunicarmos nossa solução. Esse diagrama faz parte do modelo C4 (https://c4model.com/).

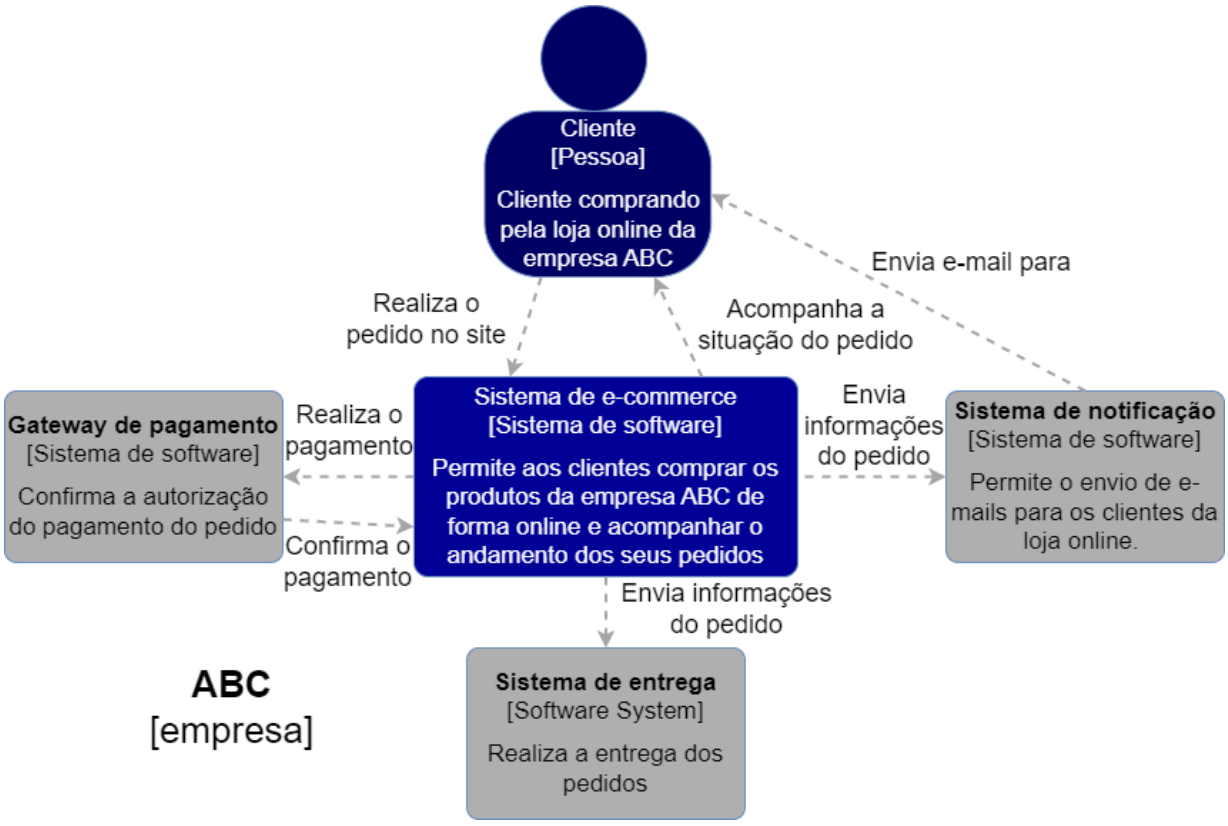

Figura 10.2: Diagrama de contexto do sistema do projeto de e-commerce.

Diagrama da arquitetura da solução

O objetivo deste diagrama é visualizar os principais componentes envolvidos na solução e os canais digitais disponíveis. Esse diagrama pode ser enriquecido com *caches*, o *datacenter* que será executado, banco de dados, protocolos de comunicação e as tecnologias utilizadas — por exemplo, o RabbitMQ na parte do *broker* de eventos, o fornecedor do *gateway* de pagamento e por aí vai.

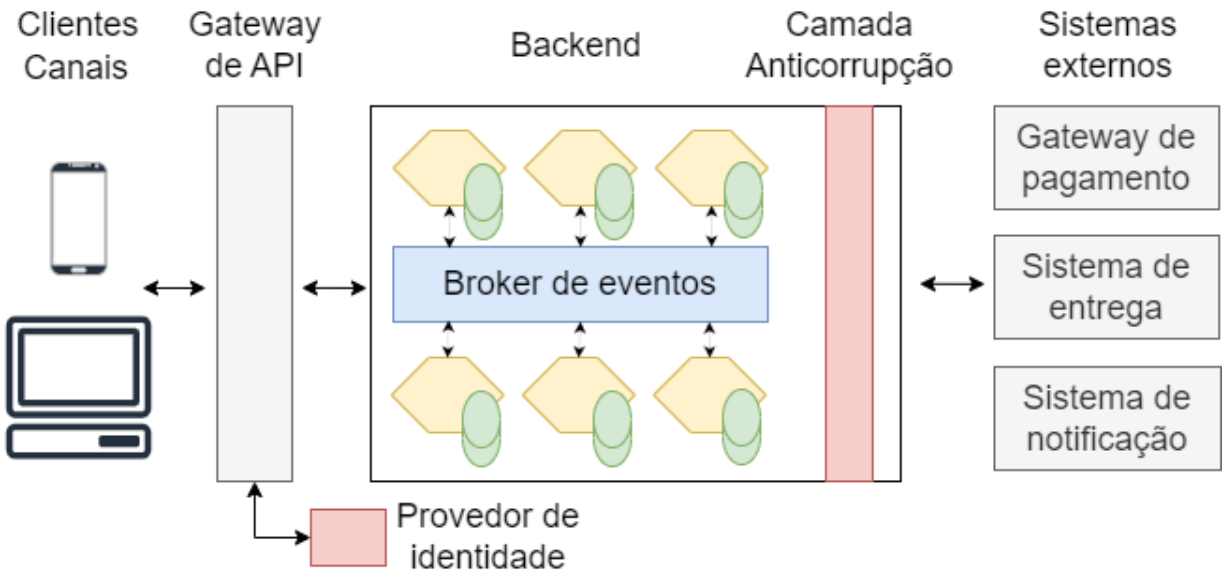

Figura 10.3: Diagrama da arquitetura da solução.

Diagrama da coreografia de eventos (The Big Picture)

O objetivo deste diagrama é ter uma visão unificada da coreografia de eventos. Com ele, conseguimos visualizar a relação entre os microsserviços e os eventos que cada um produz e consome da solução.

Imagine que, após as sessões de EventStorming, o time do projeto inicialmente definiu a solução com 9 microsserviços, sendo 3 para a camada de anticorrupção, e 20 eventos, sendo 12 eventos de negócio e 8 eventos de erro, para atender ao processo de venda da empresa ABC.

A imagem a seguir é uma versão resumida do diagrama. Para visualizar o diagrama completo, acesse-o no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/blob/main/cap10/execucao/design-solucao/diagrama-coreografia-eventos.md).

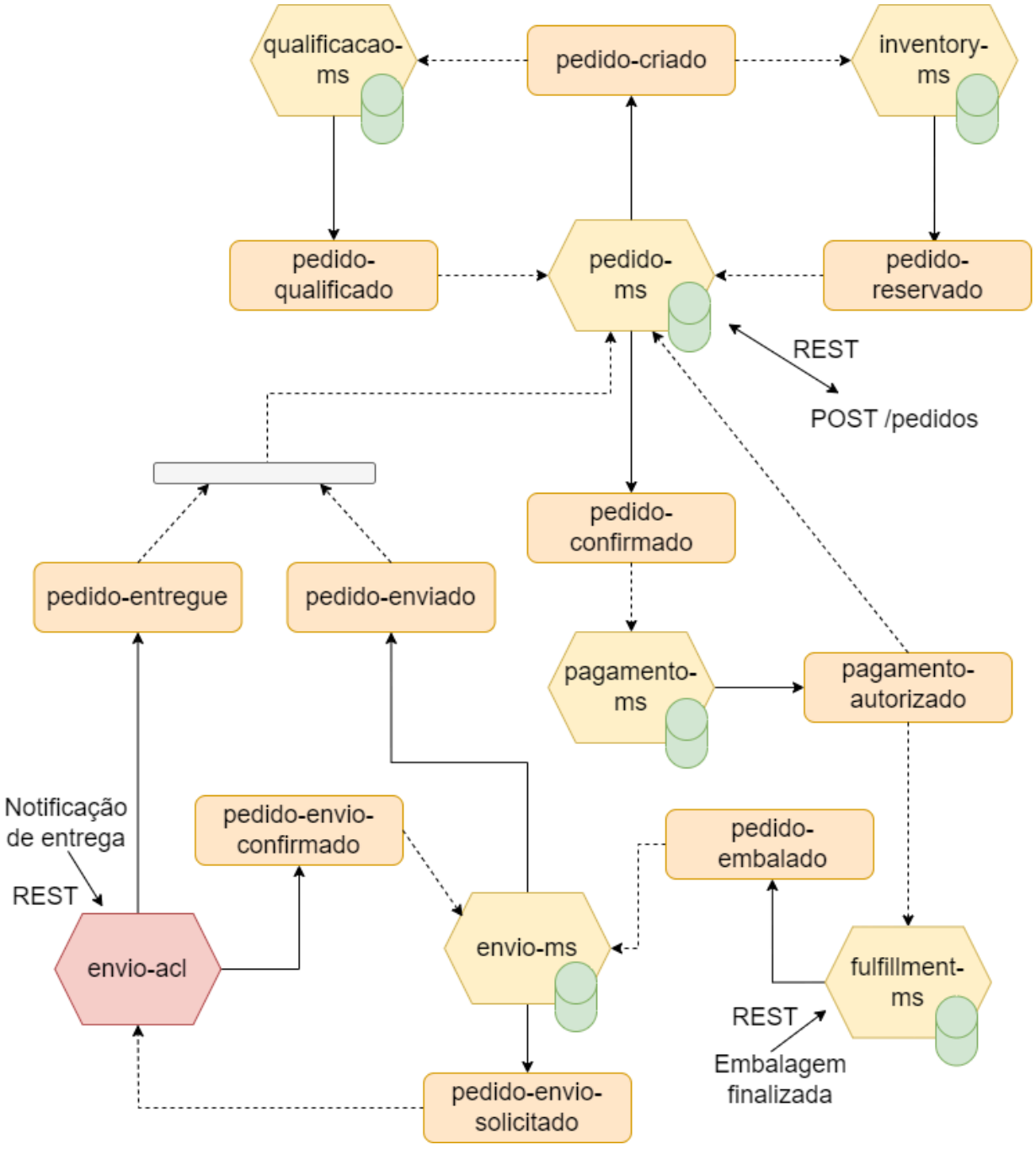

Figura 10.4: Versão resumida do diagrama da coreografia de eventos.

Diagrama de máquina de estado do pedido

O diagrama de máquina de estado tem um objetivo muito simples: mostrar graficamente a sequência de transição de estados possíveis de um objeto, no nosso caso, o `pedido`.

Na coreografia de eventos, a cada ação ou evento recebido pelo microsserviço `pedido` (dono da informação de pedido), uma transição de estado pode ocorrer. Isso faz com que consigamos acompanhar a situação atual do pedido em um determinado momento.

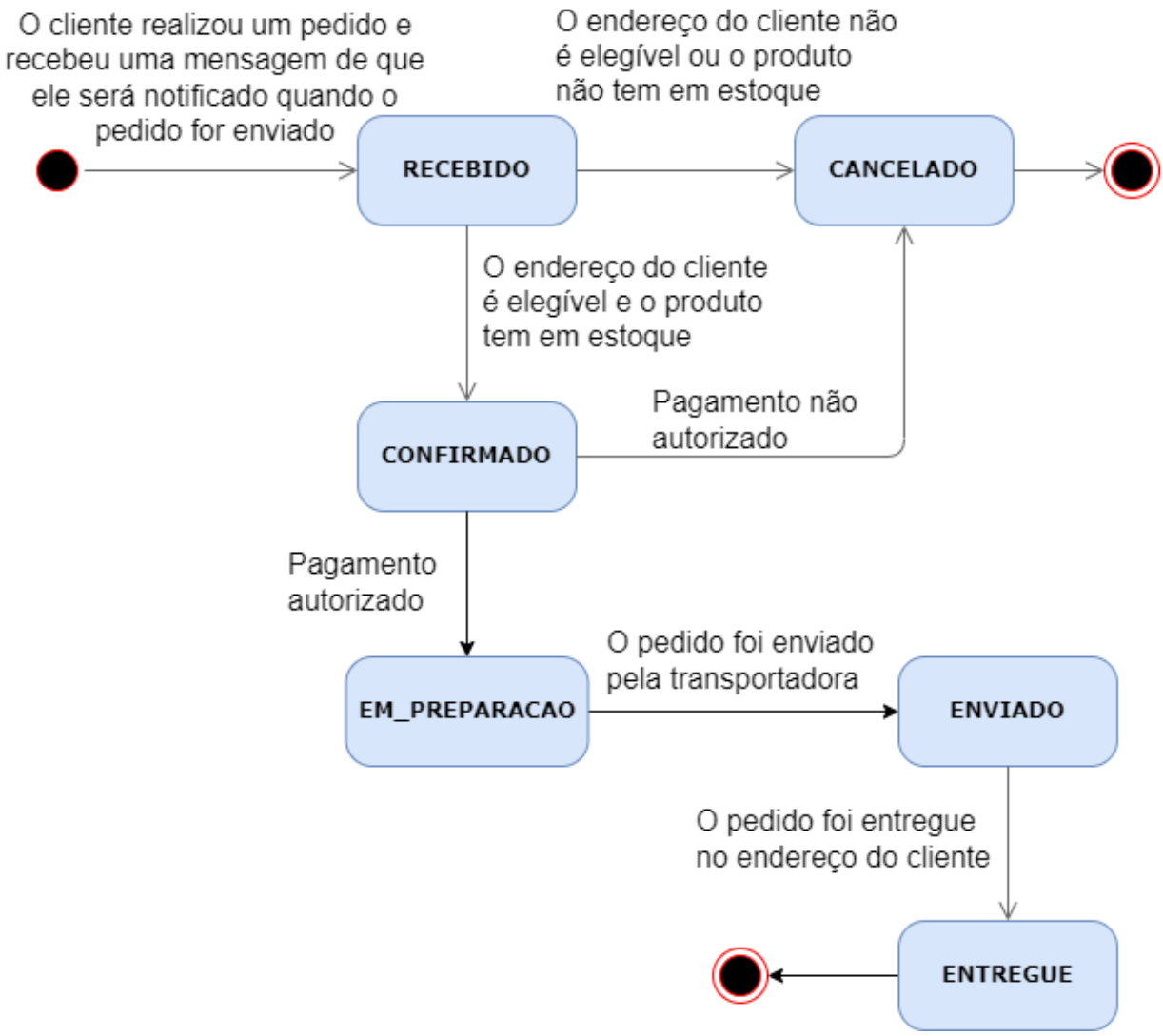

Figura 10.5: Diagrama de máquina de estado do pedido.

Diagrama de sequência do caminho feliz da compra de um produto

Apesar de a execução do fluxo de negócio através da coreografia de eventos não ser sequencial, acho que vale a pena termos esse diagrama, para nos ajudar a comunicar melhor o fluxo do caminho feliz. Esse diagrama nos ajuda a visualizar a interação entre os microsserviços, através dos eventos, para atingir um objetivo de negócio, que, no nosso caso, é a compra com sucesso de um determinado produto.

A imagem a seguir é uma versão resumida do diagrama. Para visualizar o diagrama completo, acesse-o no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/blob/main/cap10/execucao/design-solucao/diagrama-sequencia-caminho-feliz-compra-um-produto.md).

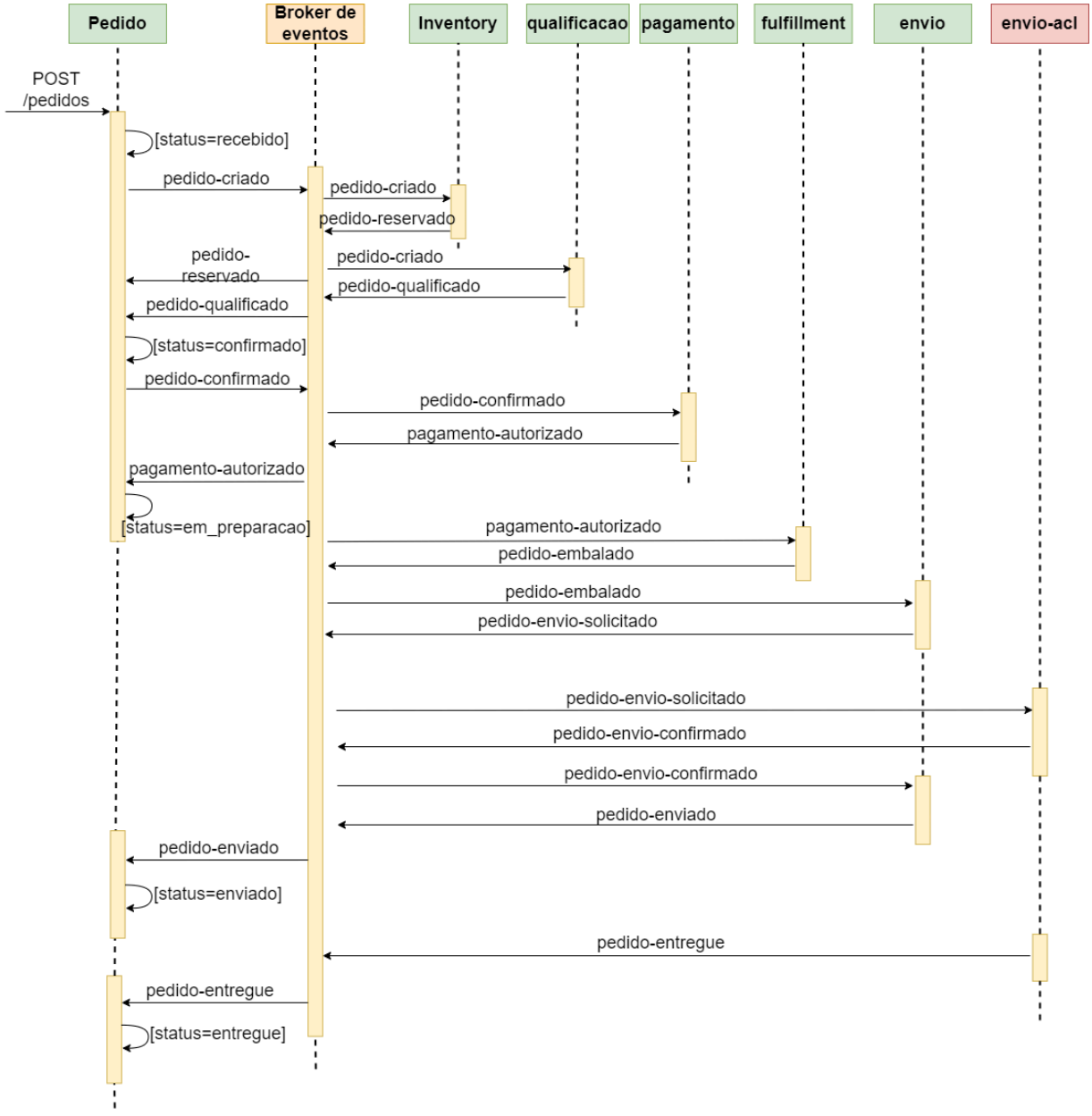

Figura 10.6: Versão resumida do diagrama de sequência do caminho feliz da compra de um produto.

OpenAPI e AsyncAPI

A pasta `contrato` no repositório do projeto tem duas pastas: `openapi` e `asyncapi`. A pasta `openapi` tem os documentos `openapi` dos microsserviços com as rotas que precisamos para executar o caminho feliz. Já a pasta `asyncapi` tem o documento com a especificação dos nossos eventos de negócio e os microsserviços que cada um produz e consome.

OpenAPI

A imagem a seguir é o *screenshot* de um trecho da documentação OpenAPI do microsserviço `pedido`:

Figura 10.7: Screenshot de um trecho da documentação OpenAPI do microsserviço pedido.

Para visualizar a documentação completa, importe os documentos OpenAPI que estão no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap10/contrato/openapi) no editor de OpenAPI (https://editor.swagger.io/).

AsyncAPI

A imagem a seguir é o *screenshot* de um trecho da documentação AsyncAPI do nosso projeto:

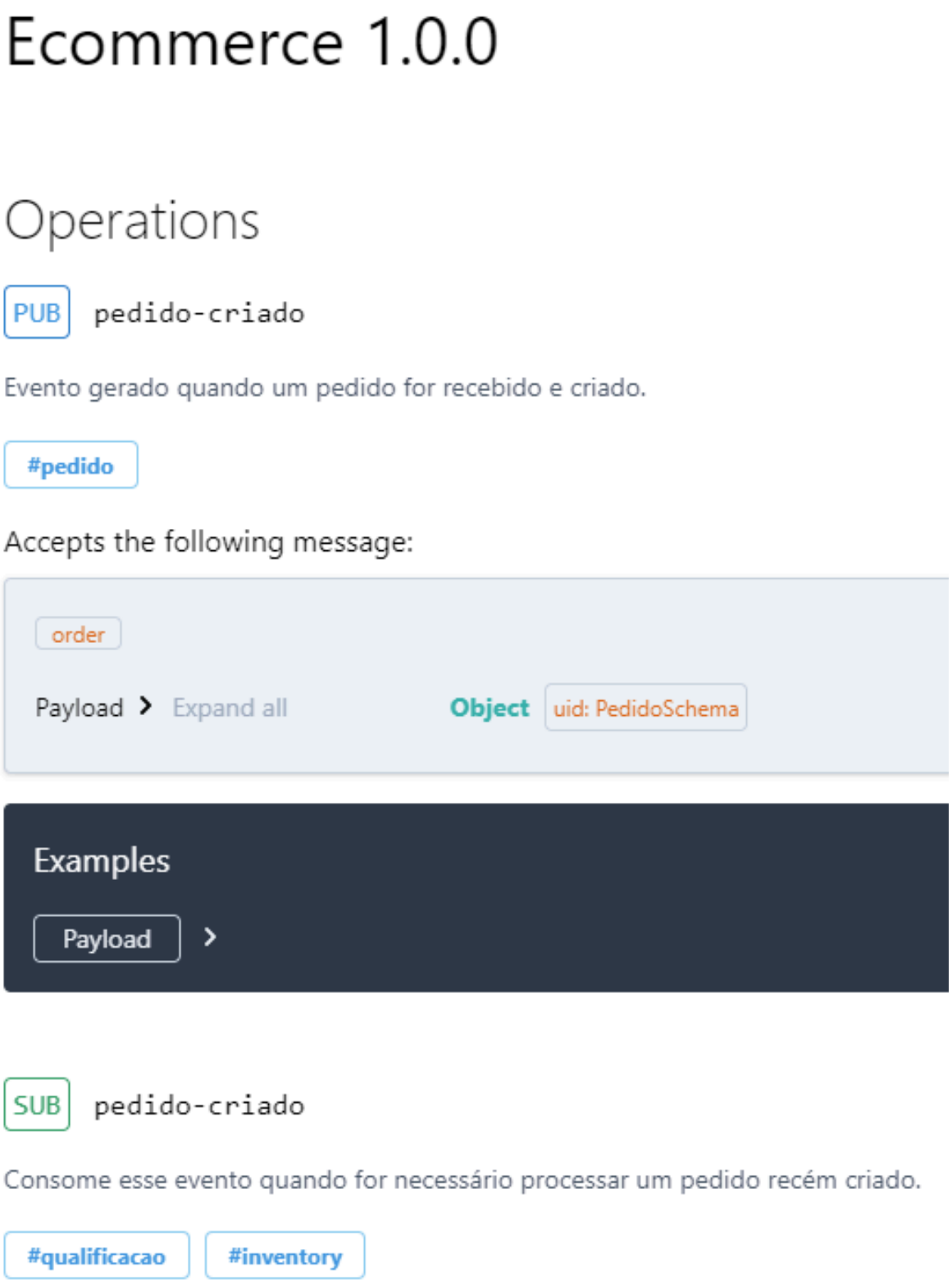

Figura 10.8: Screenshot de um trecho da documentação AsyncAPI do e-commerce da empresa ABC.

Para visualizar a documentação completa, importe o documento AsyncAPI que está no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap10/contrato/asyncapi) no AsyncAPI Studio (https://studio.asyncapi.com/).

10.5 Etapa 3: Teste

Nesta etapa, vamos definir o cenário de teste principal do caminho feliz utilizando o BDD e automatizá-lo através do *postman*. Com isso, você poderá validar a sua implementação executando o teste automatizado do cenário de teste principal do caminho feliz. Se você tiver interesse em explorar mais sobre BDD, indico o livro *BDD in Action, Second Edition*, do John Ferguson Smart e Jan Molak (2023).

Cenários de teste

Vamos explorar a funcionalidade principal que contempla o cenário de sucesso da compra de um produto no sistema de e-commerce da empresa ABC.

Nos termos do BDD, uma ***feature*** é uma parte da funcionalidade do software que ajuda os usuários ou outras partes interessadas (*stakeholders*) a atingir algum objetivo de negócio. Cada funcionalidade tem diversos cenários associados. Esses cenários são exemplos concretos de como o sistema deve se comportar em diferentes situações e podem formar a base dos critérios de aceitação da funcionalidade.

FUNCIONALIDADE: Venda on-line

A FIM DE receber todos os produtos do meu pedido na minha residência
COMO UM cliente do e-commerce da empresa ABC
EU QUERO poder realizar uma compra no site
E receber o produto na minha residência

Às vezes, é mais interessante considerar a funcionalidade não do ponto de vista do usuário final (no nosso caso, o cliente), mas da perspectiva da parte interessada (*stakeholder*), que está interessada no resultado do negócio.

FUNCIONALIDADE: Venda on-line

A FIM DE aumentar a receita e a credibilidade da empresa ABC no mercado
COMO UM gerente de venda
EU QUERO atender a todos os pedidos de compra dos meus clientes
E que todos recebam seus produtos na sua residência

CENÁRIO DE TESTE: Realizar a compra de um produto com sucesso

DADO QUE eu sou um cliente do e-commerce da empresa ABC
E eu adiciono 1 produto no carrinho de compra
E eu sigo para o *checkout*
E eu informo um endereço de entrega elegível
E eu informo dados válidos de pagamento
QUANDO eu confirmo o pedido
ENTÃO eu devo receber meu pedido com 1 produto na minha residência

Esse é o cenário de teste que vamos utilizar para validar o caminho feliz da compra de um produto feito no sistema de e-commerce da empresa ABC.

Postman

Para validar a implementação, vamos executar o teste do cenário do caminho feliz da compra de um produto através da coleção do *postman*. Na descrição do cenário de teste no nosso repositório, tem o link da pasta do *postman* com a coleção: https://github.com/rpicanco/livro-eda/tree/main/cap10/postman. Nesta pasta, existe um arquivo referente à coleção e outro arquivo referente às variáveis de ambiente. A coleção simula um pedido feito por um canal digital e valida se o pedido está em sua situação final, que é `pedido-entregue`.

10.6 Etapa 4: Implementação

Na etapa de implementação, você vai implementar a solução de acordo com os artefatos gerados na etapa de design da solução e de acordo com o cenário de teste principal do caminho feliz, criado na etapa de teste.

Você pode utilizar qualquer linguagem de programação de sua preferência. Você também pode comparar a sua solução com a solução já disponível em nosso repositório do GitHub. A solução disponível foi implementada utilizando Java (Spring Boot 3.0).

Para executar a solução disponível no nosso repositório do GitHub, baixe todos os códigos-fontes para sua máquina local e siga as instruções (https://github.com/rpicanco/livro-eda/blob/main/cap10/execucao/implementacao/passo-a-passo.md).

10.7 Sugestões de desafios extras

O objetivo desta seção é incentivar o leitor e a leitora a enriquecer o projeto com desafios extras.

- Reveja a solução adotada. Que melhorias podem ser aplicadas na solução? Por exemplo, será que faria mais sentido a verificação do endereço de entrega e a verificação do estoque fazerem parte de um processo síncrono anterior à coreografia de eventos? E o pagamento, poderíamos fazer de forma síncrona antes da coreografia dos eventos?
- Enriqueça o processo de venda e adicione mais microsserviços na solução para aumentar a complexidade e o desafio. Por exemplo, adicione funcionalidades de carrinho de compra, antifraude, *tokenização* do cartão, aplicar desconto, produto etc.;
- Adicione ao ADR a decisão e justificativa da utilização da linguagem de programação e do banco de dados de cada microsserviço, do *gateway* da API e dos fornecedores dos sistemas externos envolvidos na solução;
- Desenvolva em uma outra linguagem de programação;
- Crie uma interface gráfica para a solução;
- Defina o fluxo de compensação utilizando SAGA. Por exemplo, que fluxo de compensação devemos executar caso o pagamento do pedido não seja autorizado?
- Defina e desenvolva o processo de cancelamento do pedido;
- Defina um processo de automatização do reprocessamento dos eventos de erro;
- Adicione na solução o envio de e-mail para o cliente quando o seu pedido for entregue (avalie se não seria melhor o `pedido-ms` ser o responsável pela publicação do evento `pedido-entregue` em vez do `envio-acl` !);
- Complemente os documentos de OpenAPI adicionando novas rotas/*endpoints* e implemente-os;
- Adicione a especificação do CloudEvents no AsyncAPI;

- Crie outros cenários de teste (sucesso e erro) utilizando o modelo do BDD;
- Crie novas coleções no *postman* para os outros cenários.

Concluindo

Entender o problema e o escopo do projeto de que estamos participando, bem como os requisitos funcionais, não funcionais e as restrições, é essencial para a escolha das arquiteturas e do *broker* de eventos. Neste capítulo, definimos a arquitetura e o *broker* de eventos de um projeto simplificado de e-commerce com base no entendimento do problema que devemos resolver, nos requisitos e restrições.

O projeto está muito longe de ser um projeto real, mas ele nos dá os insumos necessários para compreender a necessidade da empresa para conseguirmos escolher as arquiteturas e o *broker* de eventos da solução. Cada projeto tem suas particularidades e necessidades. Para o nosso projeto de e-commerce simplificado, geramos alguns artefatos que são suficientes para que você consiga desenvolver a solução e enriquecê-la posteriormente com as sugestões de desafios.

No próximo capítulo, vamos simular nosso segundo projeto para a escolha da arquitetura e *broker* de eventos.

CAPÍTULO 11
Projeto 2 – Envio de notificações de eventos para parceiros

Neste nosso segundo projeto, vamos simular uma funcionalidade importante dentro de um sistema de entrega de pedidos (também conhecido como sistema de *delivery*). Precisamos notificar nossos clientes, o comerciante (*merchant*), de que o pedido do seu cliente foi entregue no endereço de entrega informado.

11.1 Entendendo o problema e definindo o escopo

A transportadora *CargaPesada* terceiriza o serviço de entrega de pedidos de diversos comerciantes, que vendem seus produtos nas suas lojas virtuais. A empresa CargaPesada tem um sistema complexo de entrega de pedidos com mais de 300 comerciantes (clientes) ativos. O escopo do nosso projeto é a parte final do processo de entrega. Precisamos notificar, via *webhook*, o sistema do comerciante, informando que o pedido do seu cliente foi entregue na sua residência.

NOMENCLATURAS

Comerciante: o cliente do sistema de entrega da empresa CargaPesada.

Cliente: o cliente do comerciante.

O processo de negócio da funcionalidade de notificação é simples:

1. O comerciante registra seu *endpoint* HTTPS (*webhook*) para ser notificado quando o pedido do seu cliente for entregue pela transportadora;
2. O comerciante solicita a entrega do pedido para o sistema de entrega. O comerciante precisa informar o número do pedido e os dados básicos do seu cliente, como CPF, nome, telefone e o endereço de entrega;
3. Uma vez que o pedido foi entregue, o sistema de entrega notifica o *endpoint* HTTPS registrado pelo comerciante confirmando a entrega do pedido. O sistema de entrega envia no corpo da chamada HTTPS algumas informações do pedido e do cliente do comerciante e o endereço de entrega do cliente;
4. O sistema de entrega tem políticas de retentativas. Ele tenta entregar a notificação ao *endpoint* HTTPS no máximo três vezes.

11.2 Requisitos funcionais, não funcionais e restrições

Baseado no processo de negócio e nas inúmeras reuniões com o time de negócios da empresa CargaPesada, chegamos ao consenso dos principais requisitos funcionais, não funcionais e das restrições impostas.

Requisitos funcionais

- Apenas um *endpoint* HTTPS público pode ser registrado por comerciante;
- O comerciante pode solicitar a entrega do pedido do seu cliente informando o número do pedido e os dados básicos do seu cliente, como CPF, nome, telefone e endereço de entrega;
- O sistema de entrega deve notificar o *endpoint* HTTPS registrado do comerciante com as seguintes informações:

    - Informações do pedido, tais como o número do pedido e o valor total;
    - Informações do cliente do comerciante, tais como CPF, nome, e-mail e telefone;
    - Endereço de entrega do cliente.
- Em caso de o *endpoint* HTTPS do comerciante estar temporariamente indisponível, o comerciante poderá contar com até três retentativas automáticas.

Requisitos não funcionais

- *Alta disponibilidade*: o sistema de entrega deve garantir um SLA de 99.999% (cinco noves) de disponibilidade no recebimento das solicitações de entrega de pedidos;
- *Escalabilidade*: o sistema de entrega deverá atender aos picos de vendas dos comerciantes;
- *Desempenho*: por se tratar de uma solução *multi-tenant* (a infraestrutura é compartilhada entre os comerciantes), o pico de venda/notificação de um comerciante não deve afetar a latência da notificação de outro comerciante. O objetivo é evitar o antipadrão do *vizinho barulhento* (*Noisy Neighbor* – https://learn.microsoft.com/pt-br/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor);
- *Latência moderada*: a notificação da entrega do pedido para o comerciante deve ter no máximo 5 minutos de atraso.

Restrições

- Como a empresa CargaPesada tem uma parceria comercial com a AWS, toda sua solução de notificação da entrega do pedido é executada na infraestrutura da AWS.

11.3 Etapa 1: Escolha do broker de eventos e das arquiteturas e plataformas

Nesta seção, vamos definir o *broker* de eventos e o padrão arquitetural de acordo com os requisitos do projeto através do registro de decisão arquitetural. Vamos criar dois registros de decisão arquitetural com a justificativa do padrão arquitetural e do *broker* de eventos que utilizaremos em nosso segundo projeto.

Título

Padrão arquitetural.

Decisão

Vamos utilizar a arquitetura *serverless* e EDA.

Justificativa

A arquitetura *serverless* e EDA possibilitam que o sistema de entrega acomode o pico de venda dos comerciantes. Os componentes da solução provisionam e desligam recursos computacionais de forma automática de acordo com a demanda, proporcionando uma melhor elasticidade na solução.

A empresa CargaPesada não precisa se preocupar com o gerenciamento e manutenção da infraestrutura para garantir a notificação, via *webhook*, ao sistema do comerciante.

Título

Broker de eventos para notificação via *webhook*.

Decisão

Amazon EventBridge. Um *broker* de eventos orientado a assinatura.

Justificativa

O Amazon EventBridge é um *broker* de eventos *serverless* orientado a assinatura da AWS. Ele tem as funcionalidades necessárias para atender ao nosso caso de uso. Com ele, conseguimos notificar *endpoints* HTTPS com políticas de retentativas e aplicar regras para fazer roteamento inteligente. Além disso, ele tem integração nativa com outros serviços da AWS, como o Amazon SQS (um *broker* orientado a fila), que é utilizado na coreografia de eventos do sistema de entrega. É através de uma fila do Amazon SQS que identificamos um pedido entregue.

O Amazon EventBridge tem suporte à semântica de *entrega pelo menos uma vez*, que é perfeita para nosso caso de uso. Garantimos a notificação ao *endpoint* HTTPS do comerciante pelo menos uma vez. Pode haver notificação duplicada. Com isso, o *endpoint* do comerciante deve ser idempotente.

Com o Amazon EventBridge atendemos à restrição imposta, que é utilizar a infraestrutura da AWS. Esse *broker* é gerenciável e escalável, atendendo aos nossos requisitos não funcionais.

11.4 Etapa 2: Design da solução

Nesta etapa, vamos gerar todos os artefatos necessários para que você possa implementar a solução.

1. Diagrama da arquitetura da solução;
2. *OpenAPI*.

> Todos os artefatos deste projeto estão disponíveis no nosso repositório do GitHub: https://github.com/rpicanco/livro-eda/tree/main/cap11.

Diagrama da arquitetura da solução

O objetivo deste diagrama é comunicar, de forma visual, os principais componentes envolvidos na solução e o fluxo da informação.

O nível de detalhe deve ser o suficiente para comunicar de forma objetiva e clara a arquitetura da solução do projeto: um diagrama muito macro pode não comunicar a arquitetura de forma efetiva e o excesso de detalhes pode poluir o diagrama e atrapalhar no entendimento do que realmente é importante.

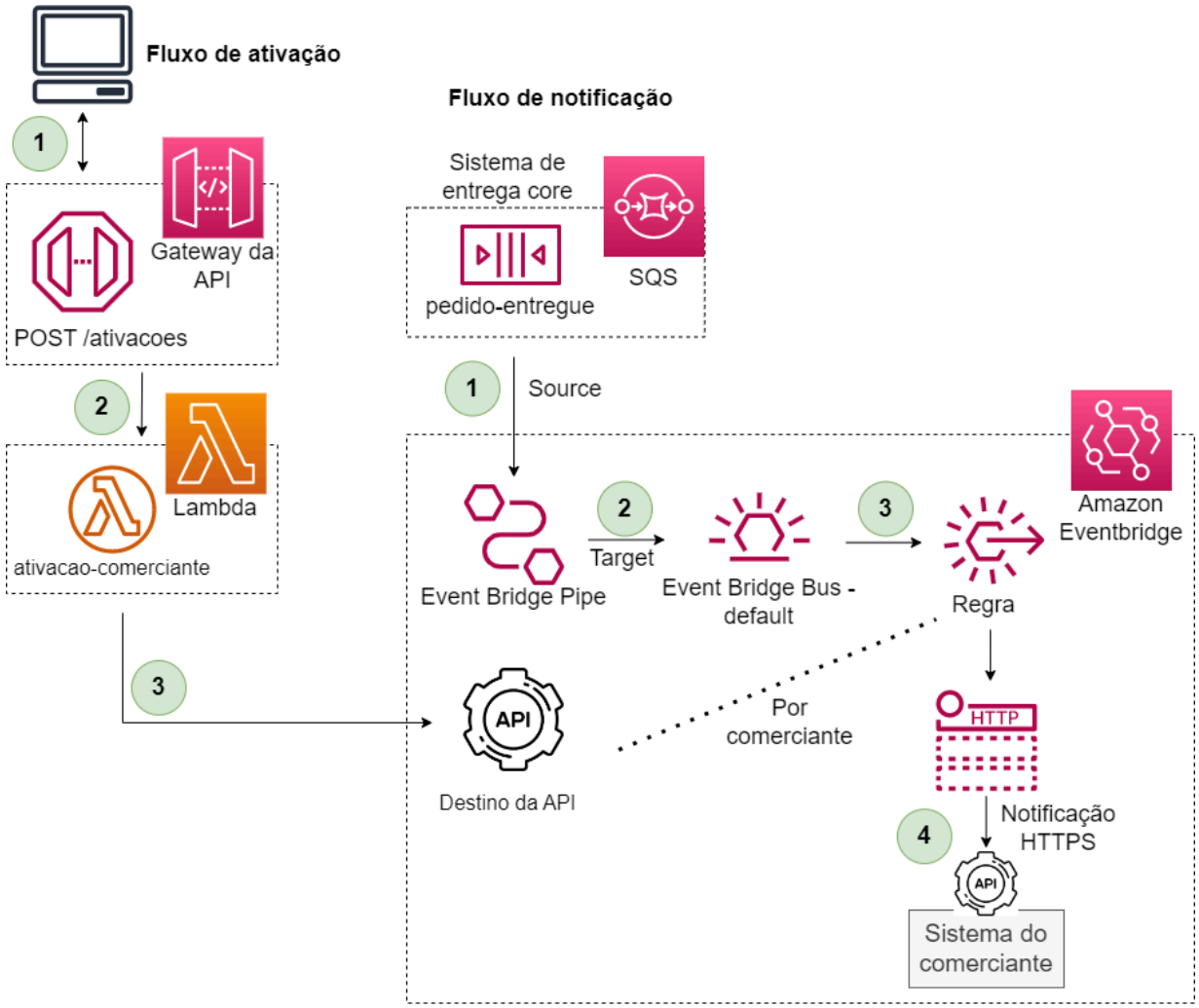

Figura 11.1: Diagrama da arquitetura da solução.

Diferentemente do nosso primeiro projeto, onde criamos o diagrama de sequência para visualizar a sequência do fluxo da informação, neste vamos descrever a sequência da informação de forma textual, conforme a numeração exposta no diagrama da arquitetura de solução.

Podemos dividir a solução em duas etapas: *fluxo de ativação*, na qual o comerciante é ativado no sistema de entrega (processo de *onboarding*); e *fluxo de notificação*, na qual o comerciante recebe, via *webhook*, as notificações de entrega de pedidos dos seus clientes.

Fluxo de ativação

1. O comerciante registra o *webhook* no sistema de entrega. O comerciante precisa informar o seu identificador e o *endpoint* `HTTPS` no corpo da solicitação, e este *endpoint* precisa estar acessível;

2. O *gateway* da API recebe a solicitação e repassa para a função lambda `ativacao-comerciante` ;

3. A função `ativacao-comerciante` cria a regra com o destino da API para o comerciante no Amazon EventBridge.

Fluxo de notificação

4. Um cliente do comerciante realiza uma compra. O sistema de entrega publica um evento `pedido-entregue` . No corpo do evento, tem o identificador do comerciante, o número do pedido e os dados básicos do seu cliente, como CPF, nome, telefone e endereço de entrega;

5. O *pipe* configurado captura o evento `pedido_entregue` e o envia para o barramento *default* do Amazon EventBridge;

6. A regra, com o destino da API configurado, do comerciante é acionada;

7. O Amazon EventBridge notifica o *endpoint* `HTTPS` registrado do comerciante.

Dois pontos importantes ao definirmos a arquitetura da nossa solução utilizando serviços de algum provedor da nuvem:

- Verifique os SLAs de disponibilidade dos serviços utilizados na solução. Os provedores disponibilizam seus SLAs para cada serviço oferecido — por exemplo, o Amazon EventBridge (https://aws.amazon.com/pt/eventbridge/sla/) e o API Gateway (https://aws.amazon.com/pt/api-gateway/sla/);
- Verifique as cotas e limites de cada serviço. Os limites desses serviços vão influenciar na definição da arquitetura da solução Por exemplo, o Amazon EventBridge tem a sua definição de limite e cotas (veja em https://docs.aws.amazon.com/pt_br/eventbridge/latest/userguide/eb-quota.html), e o Azure Event Hubs tem as suas próprias definições (https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-quotas).

OpenAPI

A imagem a seguir é um *screenshot* de um trecho da documentação *OpenAPI* do sistema de entrega:

Figura 11.2: Screenshot de um trecho da documentação OpenAPI do sistema de entrega

Para visualizar a documentação completa, importe o documento *OpenAPI* que está no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap11/contrato/openapi) no editor de OpenAPI (https://editor.swagger.io/).

11.5 Etapa 3: Teste

Nesta etapa, vamos definir o cenário de teste principal do caminho feliz utilizando o BDD e vamos automatizá-lo através do *postman*. Com isso, você poderá validar a sua implementação executando o teste automatizado do cenário de teste principal do caminho feliz.

Cenários de teste

Nesta seção, vamos explorar a funcionalidade principal com o cenário de sucesso da notificação da entrega de um pedido, via *webhook*, ao sistema do comerciante.

FUNCIONALIDADE: Notificação de entrega do pedido

A FIM DE entregar os itens dos pedidos de compra dos meus clientes
COMO UM comerciante no sistema de entrega da empresa CargaPesada
EU QUERO poder solicitar a entrega dos itens dos pedidos de compra dos meus clientes
E receber uma notificação assim que meus clientes receberem os itens dos seus pedidos de compra na sua residência

CENÁRIO DE TESTE: Notificar com sucesso a entrega do pedido para um comerciante

DADO QUE eu sou um comerciante registrado e ativo no sistema de entrega da empresa CargaPesada
E eu solicito a entrega de 1 pedido do meu cliente
E eu informo o número do pedido
E eu informo os dados básicos do meu cliente, como CPF, nome e telefone
E eu informo o endereço de entrega válido do meu cliente
QUANDO eu solicito a entrega do pedido
ENTÃO eu devo receber uma notificação no meu *endpoint* HTTPS registrado informando que o pedido do meu cliente foi entregue na sua residência

Esse é o cenário de teste que vamos utilizar para validar o caminho feliz da notificação da entrega do pedido feito pelo comerciante no sistema de entrega da empresa CargaPesada.

Postman

Para validar a implementação, vamos executar o teste do cenário do caminho feliz da notificação da entrega do pedido solicitado pelo

comerciante através da coleção do *postman*.

Na descrição do cenário de teste no nosso repositório, tem o link da pasta do *postman* com a coleção. Nesta pasta do repositório do projeto, existe um arquivo referente à coleção e outro arquivo referente às variáveis de ambiente. A coleção simula a criação do *webhook* e a notificação do pedido entregue no *endpoint* HTTPS registrado (https://github.com/rpicanco/livro-eda/tree/main/cap11/postman).

Para simular o *endpoint* HTTPS do comerciante, utilizamos o *Webhook.site* (https://webhook.site/). Na coleção do *postman*, utilizamos as APIs disponíveis do *Webhook.site* para automatizar o nosso cenário de teste: *Notificar a entrega do pedido para um comerciante com sucesso*.

11.6 Etapa 4: Implementação

Na etapa de implementação, você vai implementar a solução de acordo com os artefatos gerados na etapa de design da solução e de acordo com o cenário de teste principal do caminho feliz, criado na etapa de teste.

Você também pode comparar a sua solução com a solução já disponível em nosso repositório do GitHub. Para executar a solução, siga as instruções disponíveis em: https://github.com/rpicanco/livro-eda/blob/main/cap11/execucao/implementacao/passo-a-passo.md.

11.7 Sugestões de desafios extras

O objetivo desta seção é incentivar o leitor e a leitora a enriquecer o projeto com desafios extras.

- Adicione uma camada de autorização *OAuth2* no *gateway* da API. Apenas comerciantes autenticados e autorizados podem solicitar a entrega de pedido no sistema de entrega. O sistema não deve confiar ao comerciante que envie o seu identificador. Em posse do *token* autenticado, o *gateway* da API deve extrair o identificador do comerciante (*merchant_id*) do *token* e repassar para o back-end. O identificador do comerciante não deve fazer parte da solicitação da API definida no OpenAPI;
- Configure uma DLQ para os eventos não enviados, mesmo após todas as tentativas possíveis;
- Crie uma nova regra para comerciantes com plano *premium*. Comerciantes com esse tipo de plano têm a opção de consumir um evento no Amazon SQS para os seus pedidos entregues em vez de receber notificação a um *endpoint* HTTPS ;
- Complemente o documento de *OpenAPI* adicionado novas rotas/*endpoints* para editar o *webhook* e implemente-os;
- Gerencie e provisione toda a infraestrutura e os componentes da solução via código e de forma declarativa com o *terraform*;
- Atualize o código do fluxo de ativação para criar regras (*rules*) com no máximo três retentativas;
- Crie outros cenários de teste (sucesso e erro) utilizando o modelo do BDD;
- Crie novas coleções no *postman* para os outros cenários;
- Crie uma interface gráfica para o comerciante adicionar e editar o *webhook*.

Concluindo

Neste capítulo, definimos a arquitetura e o *broker* de eventos de um projeto de envio de notificação, via *webhook*, para o sistema do comerciante dentro de um sistema de entrega de pedidos. Essa notificação informa ao comerciante que um determinado pedido do seu cliente foi entregue na sua residência.

No próximo capítulo, vamos simular nosso terceiro e último projeto para a escolha da arquitetura e do *broker* de eventos.

Capítulo 12
Projeto 3 – Rastreamento da frota de caminhões em tempo real

Nesse nosso terceiro e último projeto, vamos propor a implementação de uma solução que permitirá o rastreamento da frota de caminhões em tempo real. Essa solução pode ser utilizada por transportadoras que desejam acompanhar sua carga em tempo real através das coordenadas geográficas dos caminhões. Isso possibilita oferecer aos seus clientes uma informação precisa sobre o tempo estimado de chegada (ETA) das suas entregas.

12.1 Entendendo o problema e definindo o escopo

A empresa fictícia *TrackerJá* oferece serviço de rastreamento da frota de caminhões em tempo real para diversas transportadoras. A TrackerJá atende atualmente 500 transportadoras ao redor do mundo com estimativa de crescimento de 40% no próximo ano. Cada transportadora varia em número de caminhões em sua frota; por exemplo, a principal transportadora que utiliza o sistema da TrackerJá tem 5 mil caminhões em sua frota.

As transportadoras precisam dos serviços da TrackerJá para monitorar sua frota em tempo real por questões de segurança, devido ao alto índice de roubo de carga, além de aprimorar o deslocamento dos caminhoneiros e estimar o tempo de entrega dos pedidos de seus clientes com mais precisão.

A funcionalidade principal do sistema de rastreamento TrackerJá é *permitir que as transportadoras consultem a localização da sua frota em tempo real*. Isso dá às transportadoras a flexibilidade para

fazerem o que quiserem com a informação da localização, como visualizar em mapas, aplicar cálculo de distância, configurar alertas quando um caminhão sai da rota planejada, gerar relatórios diversos etc.

A TrackerJá já pensa em evoluir a solução, em uma segunda fase do projeto, para atender clientes *premium*. Para esses clientes, a TrackerJá vai disponibilizar a visualização da frota da transportadora através de um mapa em vez de o próprio cliente ter sua solução de mapa.

O processo de negócio da funcionalidade de rastreamento é:

1. Cada caminhão da transportadora possui um sistema de GPS (sistema de posicionamento global), que emite um sinal da sua localização a cada 5 segundos para o sistema TrackerJá. Os dados enviados para o sistema são *o identificador* do caminhão, *as coordenadas geográficas* (latitude e longitude), a *data* e a *hora*;
2. O sistema TrackerJá agrega as informações da localização dos caminhões com os nomes dos motoristas que estão dirigindo o caminhão. As informações do motorista (*nome*) são mantidas em uma base de dados associada ao caminhão;
3. Com as informações agregadas, o sistema TrackerJá disponibiliza uma robusta solução de consulta para que as transportadoras busquem a localização da sua frota em tempo real.

12.2 Requisitos funcionais, não funcionais e restrições

Com base no processo de negócio e nas inúmeras reuniões com o time de negócios da empresa TrackerJá, chegamos ao consenso dos

principais requisitos funcionais, não funcionais e das restrições impostas.

Requisitos funcionais

- O sistema de GPS em cada caminhão deve enviar para o sistema TrackerJá o identificador do caminhão, as coordenadas geográficas (latitude e longitude), a data e hora a cada 5 segundos;
- A base de dados da alocação dos motoristas tem apenas informações do nome do motorista e o identificador do caminhão em que ele está alocado na viagem;
- As transportadoras (*clientes*) podem consultar, via API, a localização de toda a sua frota em tempo real a qualquer momento;
- As transportadoras podem reproduzir uma viagem específica através da API.

Para a segunda fase do projeto (visualizar em um mapa), para atender aos clientes *premium*, já sabemos os seguintes requisitos funcionais:

- Os clientes podem acompanhar a localização atual de toda a sua frota em tempo real em um mapa;
- Os clientes podem reproduzir e visualizar no mapa uma viagem específica em uma data escolhida.

Requisitos não funcionais

- *Alta disponibilidade*: o sistema deve garantir que receberá todas as informações da localização dos caminhões a cada 5 segundos e que atenderá a todas as consultas por localização das transportadoras;
- *Escalabilidade*: o sistema deve garantir que atenderá a todas as consultas por localização em tempo real das transportadoras, independente da quantidade de caminhões da sua frota;

- *Baixa latência*: o sistema precisa ter uma baixa latência na consulta (*query*). A estimativa calculada, baseada na quantidade média de caminhões por transportadora, é de 56 mil QPS (*query per second*) na consulta da localização por caminhão.

ESTIMATIVA DO QPS (*QUERY PER SECOND*, OU CONSULTA POR SEGUNDO)

Foi utilizado um cálculo aproximado chamado *Back-of-the-envelope* (https://bytebytego.com/courses/system-design-interview/back-of-the-envelope-estimation), no qual:

- Temos 700 transportadoras (500 + 40% estimado para o próximo ano);
- Cada transportadora tem, em média, 400 caminhões na sua frota;
- Então: 700 x 400 = 280 mil. Vamos utilizar 280 mil caminhões na nossa estimativa.

Uma transportadora faz 1 consulta (*query*) a cada 5 segundos por caminhão (pois é o intervalo de tempo de envio da localização do GPS para a TrackerJá). Sendo assim, uma transportadora pode fazer 17.280 consultas (*queries*) por dia por caminhão.

```
280.000 X 17.280 / 86.400 (quantidade de segundos em um dia) = 56.000 QPS
```

Vamos utilizar 56 mil QPS na nossa estimativa.

12.3 Etapa 1: Escolha do broker de eventos e das arquiteturas e plataformas

Nesta etapa, vamos definir o *broker* de eventos, o padrão arquitetural e a solução OLAP (*Online Analytical Processing*) para atender ao nosso requisito de baixa latência nas consultas das localizações. Vamos criar três registros de decisão arquitetural com a justificativa do padrão arquitetural, do *broker* de eventos e da solução OLAP que utilizaremos em nosso terceiro e último projeto.

Registro de decisão arquitetural

Título

Padrão arquitetural.

Decisão

Vamos utilizar a arquitetura de *streaming* de eventos.

Justificativa

O envio da localização dos caminhões pelo GPS a cada 5 segundos são fluxos contínuos de eventos de série (*stream*). A arquitetura de *streaming* de eventos nos permite capturar, armazenar, processar e enviar esses eventos para diferentes tecnologias de destino em uma sequência ordenada de eventos que são continuamente anexadas (*append-only*) no final do tópico ou log. Esses eventos podem ser reproduzidos na sequência a partir de qualquer posição do log.

Título

Broker de eventos para processamento de *streaming*.

Decisão

Confluent Apache Kafka. Um *broker* de eventos orientado a log.

Justificativa

O Apache Kafka, da Confluent, nos dá uma solução completa de uma plataforma de *streaming* de eventos na nuvem com suporte à semântica de *entrega pelo menos uma vez*. A plataforma tem funcionalidades que nos permitem receber as informações da localização dos caminhões, juntamente com informações adicionais em banco de dados diversos através de conectores, para armazená-las, agregá-las e processá-las em tempo real, utilizando uma solução distribuída, escalável e tolerante a falhas.

Uma das principais características desse tipo de *broker* é sua capacidade de reter e reproduzir eventos a partir de qualquer posição do log. Essa característica atende à necessidade da TrackerJá de reproduzir uma viagem específica em uma data escolhida.

Título

Consulta da localização da frota de caminhões em tempo real.

Decisão

Apache Pinot. Uma solução OLAP.

Justificativa

O Apache Pinot (https://pinot.apache.org/) é uma solução de armazenamento de dados OLAP distribuído em tempo real. Ele é projetado para responder a consultas OLAP com baixa latência. É ideal para análises em tempo real voltadas para o usuário (*user-facing*), no nosso caso, transportadoras. Sua solução é baseada em armazenamento colunar, com técnicas de indexação e pré-agregação.

Ele foi originalmente desenvolvido em 2014 pelos engenheiros do LinkedIn e Uber para resolver os casos de uso de análise em tempo real de missão crítica. O Apache Pinot é utilizado pelo LinkedIn para

atender funcionalidades de impressões, quem visualizou meu perfil, análise de artigos e *talent insights*. Ele atende a mais de 50 produtos *user-facing* do LinkedIn, ingerindo milhões de eventos por segundo e atendendo a mais de 100 mil consultas por segundo (QPS) com latência de milissegundos.

No Kafka Summit de 2021 (https://www.confluent.io/events/kafka-summit-europe-2021/analyzing-petabyte-scale-financial-data-with-apache-pinot-and-apache-kafka/), a Stripe divulgou seu experimento com Apache Pinot, chegando a 200 mil consultas por segundo (QPS).

O Apache Pinot oferece API de consulta, que vamos utilizar para permitir que as transportadoras consultem a localização dos seus caminhões em tempo real no sistema da TrackerJá.

12.4 Etapa 2: Design da solução

Nesta etapa, vamos gerar apenas o diagrama da arquitetura da solução para que você possa implementar a solução.

> Todos os artefatos desse projeto estão disponíveis no nosso repositório do GitHub (https://github.com/rpicanco/livro-eda/tree/main/cap12).

Diagrama da arquitetura da solução

O objetivo deste diagrama é comunicar, de forma visual, os principais componentes envolvidos na solução e o fluxo da informação.

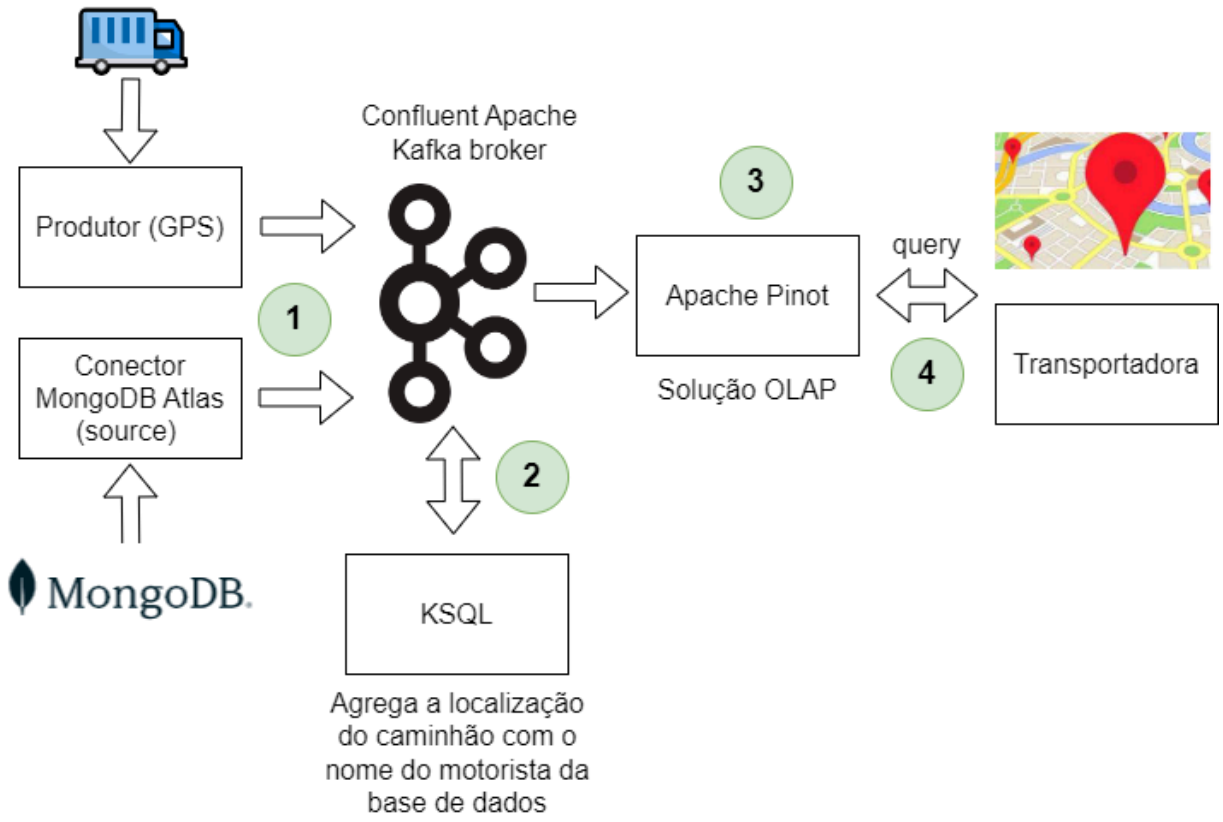

Figura 12.1: Diagrama da arquitetura da solução.

Como no projeto anterior, vamos descrever a sequência da informação de forma textual, conforme a numeração exposta no diagrama da arquitetura de solução:

1. O *broker* recebe informações da localização dos caminhões através do GPS a cada 5 segundos, além de informações do nome do motorista que está dirigindo o caminhão naquele momento. O *broker* recebe no tópico `localizacoes` a localização do caminhão em tempo real e, no tópico `db.trackerja-db.veiculos`, o nome do motorista com a identificação do caminhão;
2. Com o *KSQL*, agregamos as informações da localização do caminhão com o nome do motorista e publicamos no tópico `localizacoes_enriquecidas`;
3. O Apache Pinot tem suporte de ingestão de dados em tempo real nativo para o Apache Kafka, por isso não precisamos de um

conector. O Apache Pinot é alimentado pelos eventos do tópico `localizacoes_enriquecidas` ;

4. As transportadoras utilizam a API de consulta do Apache Pinot para buscar as informações da localização em tempo real dos caminhões da sua frota. Para a segunda fase do projeto, um mapa será alimentado automaticamente para que as transportadoras visualizem sua frota diretamente no mapa.

A seguir, a imagem com a estrutura de exemplo dos três tópicos envolvidos na solução:

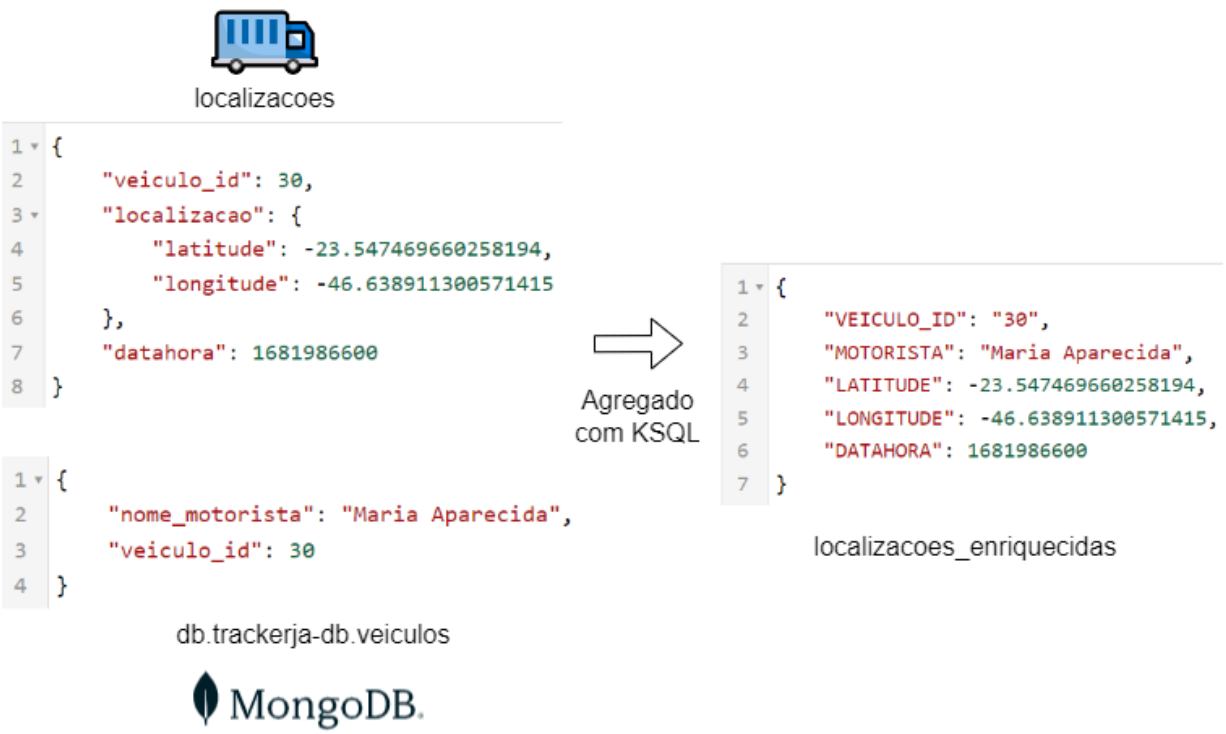

Figura 12.2: Conteúdo dos tópicos envolvidos da solução.

12.5 Etapa 3: Teste

Nesta etapa, vamos definir o cenário de teste principal do caminho feliz utilizando o BDD e vamos automatizá-lo através do *postman*.

Com isso, você poderá validar a sua implementação executando o teste automatizado do cenário de teste principal do caminho feliz.

Cenários de teste

Nesta seção, vamos explorar a funcionalidade principal com o cenário de sucesso da consulta da rota percorrida de um determinado caminhão.

Funcionalidade: Consulta da rota percorrida da minha frota de caminhões

A FIM DE obter a rota percorrida da minha frota de caminhões
COMO UM(A) transportadora no sistema de rastreamento da empresa TrackerJá
EU QUERO poder consultar a rota percorrida de qualquer caminhão da minha frota

Cenário de teste: Consultar a rota percorrida de um determinado caminhão da minha frota com sucesso

DADO QUE eu sou uma transportadora válida no sistema de rastreamento da empresa TrackerJá
E o caminhão da minha frota com número de identificação *10* saiu às 06h00 do dia 20 de abril do Rio de Janeiro
E o caminhão da minha frota com número de identificação *10* chegou às 08h55 do dia 20 de abril em São Paulo
E o motorista do caminhão com número de identificação *10* é o *João da Silva*
QUANDO eu solicito a rota percorrida do meu caminhão com número de identificação *10*
ENTÃO eu devo receber as localizações (latitude e longitude) da rota percorrida pelo caminhão com número de identificação *10* entre o Rio de Janeiro e São Paulo
E o horário de saída do Rio de janeiro foi às 06h00 do dia 20 de abril
E o horário de chegada em São Paulo foi às 08h55 do dia 20 de abril
E eu devo receber o nome do motorista *João da Silva*

Esse é o cenário de teste que vamos utilizar para validar o caminho feliz da consulta da rota percorrida de um determinado caminhão no sistema de rastreamento da empresa TrackerJá.

Vamos utilizar dados de localização e data/hora *fakes*. Mas a solução seria a mesma, como se fosse enviado o sinal do GPS a cada 5 segundos. O importante é validar se as localizações (latitude e longitude) e o nome do motorista enviados na simulação são os mesmos recuperados na API de consulta do Apache Pinot.

Postman

Para validar a implementação, vamos executar o teste do cenário do caminho feliz da consulta da rota percorrida de um determinado caminhão através da coleção do *postman*.

Na descrição do cenário de teste no nosso repositório, tem o link da pasta do *postman* com a coleção. Nesta pasta no repositório do projeto, existe um arquivo referente à coleção e outro arquivo referente às variáveis de ambiente. A coleção é referente ao cenário do caminho feliz da consulta da rota percorrida de um determinado caminhão (https://github.com/rpicanco/livro-eda/blob/main/cap12/postman).

12.6 Etapa 4: Implementação

Na etapa de implementação, você vai implementar a solução de acordo com os artefatos gerados na etapa de design da solução e de acordo com o cenário de teste principal do caminho feliz, criado na etapa de teste.

Você também pode comparar a sua solução com a solução já disponível em nosso repositório do GitHub. Para executar a solução, siga as instruções em: https://github.com/rpicanco/livro-eda/blob/main/cap12/execucao/implementacao/passo-a-passo.md.

12.7 Sugestões de desafios extras

O objetivo desta seção é incentivar o leitor e a leitora a enriquecer o projeto com desafios extras.

- Para evitarmos que uma transportadora consulte as localizações de caminhões de outra transportadora em uma solução *multi-tenant*, crie uma terceira base de dados com as informações do

identificador do veículo e o identificador da transportadora e envie para o Apache Kafka em um novo tópico. Agregue com o KSQL o identificador da transportadora no tópico `localizacoes_enriquecidas`;

- Aprimore a validação do teste no *postman* para validar a cidade de saída e de chegada de um determinado caminhão de acordo com a latitude e longitude capturadas do GPS. *DICA*: utilize a API do Google de geocodificação reversa (*reverse geocoding*);
- Não permita que a transportadora consulte diretamente na API do Apache Pinot. Crie um OpenAPI com a API de consulta da TrackerJá e adicione na solução um *gateway* de API;
- Adicione uma camada de autorização *OAuth2* no *gateway* da API. Apenas transportadoras autenticadas e autorizadas podem buscar a localização dos caminhões da sua frota. O sistema não deve confiar à transportadora que envie o seu identificador. Em posse do *token* autenticado, o *gateway* da API deve extrair o identificador da transportadora (*transportadora_id*) do *token* e considerar na busca. O identificador da transportadora não deve fazer parte da solicitação da API definida no OpenAPI;
- Implemente a segunda fase do projeto. Adicione a visualização da localização da frota em um mapa com *refresh* (chamada à API de consulta) a cada 10 segundos;
- Adicione cache na consulta da localização pelo API Gateway de acordo com o intervalo definido do *refresh*;
- Faça uma nova estimativa aproximada do QPS com a consulta a cada 10 segundos em vez de 5 segundos;
- Crie um conector *ElasticSearch Sink* para enviar as localizações enriquecidas para o ElasticSearch e faça a busca por lá;
- Use o Kafka Stream para fazer a agregação das informações em vez de usar o KSQL;
- Utilize outra solução de *streaming* de dados no projeto — por exemplo, o Amazon Kinesis (https://aws.amazon.com/pt/kinesis/);
- Enriqueça a simulação do GPS enviando dados mais realistas de localização e data/hora.

Visualizando a rota percorrida de um caminhão no mapa

Antes de finalizar o capítulo, vamos visualizar a rota de um determinado caminhão recuperado no Apache Pinot no mapa. Vamos baixar o resultado da pesquisa no Apache Pinot no formato `csv` e manualmente carregar no mapa do Kepler (https://kepler.gl/). Vamos também criar um efeito para simular o caminhão em movimento na rota *Rio de Janeiro x São Paulo* em tempo real.

Para importar e visualizar a rota de um caminhão no mapa, siga o passo a passo que está disponível no nosso projeto do GitHub (https://github.com/rpicanco/livro-eda/blob/main/cap12/execucao/implementacao/visualizando-a-rota-percorrida-de-um-caminhao-no-mapa.md).

Concluindo

Neste capítulo, definimos a arquitetura, o *broker* de eventos e a solução OLAP de um projeto de rastreamento da frota de caminhões em tempo real. Cada caminhão emite um sinal da sua localização para o sistema de rastreamento. O sistema processa os fluxos contínuos de eventos de série (*stream*) e os disponibiliza para uma solução OLAP, projetada para responder a consultas das localizações com baixa latência.

Para finalizar, carregamos manualmente o resultado da busca da rota de um determinado caminhão em um mapa para visualizar a rota percorrida e simulamos o mapa sendo atualizado em tempo real.

Chegamos ao final do livro!!

Gostaria de agradecê-lo(a) por ter lido este livro. Foi um trabalho de muitos meses, cansativo, porém prazeroso, poder compartilhar, através do livro, o que vivi e aprendi nesses anos. Espero que tenha gostado e que ele contribua na implementação dos seus projetos de EDA.

Um abraço.
Roberto Pinto Picanço

Capítulo 13
Bibliografia

APACHE SOFTWARE FOUNDATION. (Site institucional). Apache Kafka. c2023. Disponível em: https://kafka.apache.org/. Acesso em: 15 jun. 2023.

AMAZON WEB SERVICES. (Canal). AWS re:Invent 2014 | Announcing AWS Lambda. *Youtube*, 13 nov. 2014. Disponível em: https://www.youtube.com/watch?v=9eHoyUVo-yg. Acesso em: 15 jun. 2023.

AWS. *O que é uma arquitetura orientada por eventos?* c2023a. Disponível em: https://aws.amazon.com/pt/event-driven-architecture/. Acesso em: 15 jun. 2023.

AWS. *Amazon EventBridge*. c2023b. Disponível em: https://aws.amazon.com/pt/eventbridge/. Acesso em: 15 jun. 2023.

AWS. *Amazon Kinesis*. c2023c. Disponível em: https://aws.amazon.com/pt/kinesis/. Acesso em: 15 jun. 2023.

AWS. *Amazon Managed Streaming for Apache Kafka*

(MSK). c2023d. Disponível em: https://aws.amazon.com/pt/msk/. Acesso em: 15 jun. 2023.

AWS. *Amazon MQ*. c2023e. Disponível em: https://aws.amazon.com/pt/amazon-mq/. Acesso em: 15 jun. 2023.

AWS. *Amazon SNS*. c2023f. Disponível em: https://aws.amazon.com/pt/sns/. Acesso em: 15 jun. 2023.

AWS. *Amazon SQS*. c2023g. Disponível em: https://aws.amazon.com/pt/sqs/. Acesso em: 15 jun. 2023.

BELLEMARE, Adam. *Building Event-Driven Microservices*. Boston: O'Reilly Media, 2020.

CONFLUENT. (Página institucional). c2023. Disponível em: https://www.confluent.io/. Acesso em: 15 jun. 2023.

EVENTSTORMING. (Página institucional). Disponível em: https://www.eventstorming.com/. Acesso em: 15 jun. 2023.

FORRESTER. *Predictions 2022*: Software Development. 2021. Disponível em: https://www.forrester.com/report/predictions-2022-software-development/RES176485. Acesso em: 15 jun. 2022.

FORRESTER. *Predictions 2023*: Software Development. 2022. Disponível em: https://www.forrester.com/report/predictions-2023-software-development/RES178160. Acesso em: 15 jun. 2023.

FOWLER, Martin. Focusing on Events. *martinFowler.com*, 2006. Disponível em: https://martinfowler.com/eaaDev/EventNarrative.html. Acesso em: 20 jul. 2023.

VINCENT, Paul; GUPTA, Aashish. Embrace Event-Driven Architecture With Responsive Decisions for Digital Business Solutions. *Gartner*, 2017. Disponível em: https://www.gartner.com/document/code/329513?ref=grbody&refval=3732617. Acesso em: 15 jun. 2023.

GARTNER. *Maturity Model For Event Driven Architecture*. 2019. Disponível em:

https://www.gartner.com/en/documents/3942102. Acesso em: 15 jun. 2023.

GARTNER. *Gartner Predicts 2022*: Modernizing Software Development is Key to Digital Transformation. 2021a. Disponível em: https://www.gartner.com/en/documents/4009915. Acesso em: 15 jun. 2023.

GARTNER. How to Identify Your Event-Driven Architecture Use Cases to Select the Best-Fit Event Broker. 2021b. Disponível em: https://www.gartner.com/. Acesso em: 15 jun. 2023.

GARTNER. *Event-driven Architecture (EDA)*. c2023. Disponível em: https://www.gartner.com/en/information-technology/glossary/eda-event-driven-architecture. Acesso em: 15 jun. 2023.

GOOGLE. Arquiteturas orientadas a eventos. *Google Cloud*, c2022b. Disponível em: https://cloud.google.com/eventarc/docs/event-driven-architectures. Acesso em: 15 jun. 2023.

GOOGLE. Pub/Sub. *Google Cloud*, c2022b. Disponível em: https://cloud.google.com/pubsub. Acesso em:

15 jun. 2023.

IBM. *IBM MQ*. c2023. Disponível em: https://www.ibm.com/br-pt/products/mq. Acesso em: 15 jun. 2023.

KAFKA. Some Common Philosophical Questions. *In*: *Kafka Protocol Guide*. c2023. Disponível em: https://kafka.apache.org/0100/protocol.html#protocol_philosophy. Acesso em: 15 jun. 2023.

KLEPPMANN, Martin. *Designing Data-Intensive Applications*: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Boston: O'Reilly Media, 2017.

MICROSOFT. Event-driven architecture style. *Microsoft Learn*, c2023a. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven. Acesso em: 15 jun. 2023.

MICROSOFT. Grade de eventos. *Microsoft Azure*, c2023a. Disponível em: https://azure.microsoft.com/pt-br/products/event-grid. Acesso em: 15 jun. 2023.

MICROSOFT. Hubs de eventos. *Microsoft Azure*, c2023b. Disponível em: https://azure.microsoft.com/pt-br/products/event-hubs. Acesso em: 15 jun. 2023.

MICROSOFT. Barramento de serviço. *Microsoft Azure*, c2023c. Disponível em: https://azure.microsoft.com/pt-br/products/service-bus. Acesso em: 15 jun. 2023.

NEIL, Avery. Journey to Event Driven – Part 1: Why Event-First Programming Changes Everything. *Confluent*, 2019. Disponível em: https://www.confluent.io/blog/journey-to-event-driven-part-1-why-event-first-thinking-changes-everything/. Acesso em: 15 jun. 2023.

NEWMAN, Sam. *Building Microservices*: Designing Fine-Grained Systems. 2. ed. Boston: O'Reilly Media, 2021.

OLLIFFE, Gary. Choosing Event Brokers: The Foundation of Your Event-Driven Architecture. *Gartner*, 2020. Disponível em: https://www.gartner.com/en/documents/3986571. Acesso em: 15 jun. 2023.

RED HAT. *O que é arquitetura orientada a eventos?*. 2019. Disponível em: https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture. Acesso em: 15 jun. 2023.

RICHARDSON, Chris. *Microservices Patterns*: With Examples in Java. Nova Iorque: Manning, 2018.

RIEL, Rey. Event-Driven Architecture vs. Event-Driven Integration: What's the Difference? *Solace*, 2022. Disponível em: https://solace.com/blog/event-driven-architecture-difference-event-driven-integration/. Acesso em: 15 jun. 2023.

ROCHA, Hugo F. O. *Practical Event-Driven Microservices Architecture:* Building Sustainable and Highly Scalable Event-Driven Microservices. Nova Iorque: Apress, 2021.

SBARSKI, Peter; CUI, Yan; NAIR, Ajay. *Serverless Architectures on AWS*. 2. ed. Nova Iorque: Manning, 2022.

SCOTT, Dylan; GAMOV, Viktor; KLEIN, Dave. *Kafka in Action*. Nova Iorque: Manning, 2022.

SENSEDIA. *Event Driven Reference Guide*. 2021.

Disponível em: https://content.sensedia.com/pt-br/event-driven-0. Acesso em: 5 set. 2023.

SENSEDIA. *Events Hub*. c2023. Disponível em: https://www.sensedia.com.br/produtos/events-hub. Acesso em: 15 jun. 2023.

SHAPIRA, Gwen; PALINO, Todd; SIVARAM, Rajini; PETTY, Krit. *Kafka:* The Definitive Guide: Real-Time Data and Stream Processing at Scale. 2. ed. Boston: O'Reilly Media, 2021.

SHRIVASTAVA, Saurabh; SRIVASTAV, Neelanjali. *Solutions Architect's Handbook*: Kick-start your career as a solutions architect by learning architecture design principles and strategies. Birmingham: Packt Publishing Co., 2022.

SMART, John F.; MOLAK, Jan. *BDD in Action*: Behavior-Driven Development for the whole software lifecycle. 2. ed. Nova Iorque: Manning, 2023.

SOLACE. *Getting in Sync*: Unlocking the Exponential Business Value of Real-Time Event-Driven Data Flows. c2021. Disponível em: https://solace.com/resources/white-papers/wp-download-idc-eda-survey. Acesso em: 15 jun. 2023.

SOLACE. *The Great EDA Migration*: 2021 Survey Results. 2021. Disponível em: https://solace.com/resources/solace-event-driven-architecture-resources/wp-download-the-great-eda-migration-2021-survey-results. Acesso em: 15 jun. 2023.

SOLACE. *PubSub+ Event Broker*. c2023. Disponível em: https://solace.com/pt/products/event-broker/. Acesso em: 15 jun. 2023.

THOO, Eric; GUTTRIDGE, Keith; BHULLAR, Bindi; PILLAI, Shameen; SINGH, Abhishek. Gartner Magic Quadrant for Enterprise Integration Platform as a Service. *Gartner*, 2021. Disponível em: https://www.gartner.com/en/documents/4006317. Acesso em: 15 jun. 2023.

RABBITMQ. (Site institucional). c2023. Disponível em: https://www.rabbitmq.com/. Acesso em: 15 jun. 2023.

VOLATILIDADE, incerteza, complexidade e ambiguidade. *In*: WIKIPÉDIA, a enciclopédia livre. *Wikimedia*, c2023. Disponível em: https://pt.wikipedia.org/wiki/Volatilidade,_incerteza,_

complexidade_e_ambiguidade. Acesso em: 20 jul. 2023.

XU, Alex; LAM, Sahn. *System Design Interview* – An Insider's Guide: Volume 2. Santa Clara, CA: ByteByteGo (Byte Code LLC), 2022.